The Code Orchestrator’s Guide to Terminology and Practice Standards

How a non-programmer can direct OpenAI Codex like a competent technical lead

Report edition: May 6, 2026

Table of Contents

Major parts and key sections. Page numbers are based on the rendered Word document.

Source boundary and labels **3**
Part 1: What Code Orchestration Is **4**
Roles: who does what 4
Why orchestration is not vibe coding 5
The Dangerous Beginner Fantasy: “The Agent Will Just Figure It Out” 5
Part 2: The Mental Model of a Codebase **6**
Part 3: How Codex Should Be Directed **15**
Taxonomy of Codex task types 16
Part 4: Prompting Standards for Codex **18**
Reusable serious Codex prompt template 18
Why vague prompts cause bad outcomes 21
Part 5: Software Design Concepts Every Orchestrator Needs **21**
Part 6: Testing and Validation **27**
How to make Codex prove work 28
The Agent’s Favorite Lie: “This Should Work” 28
Part 7: Debugging as a Discipline **29**
Debugging workflow 33
Part 8: Version Control, Git, and Worktrees **34**
Practical Git standards for agent work 39
Part 9: Agentic Workflow Design **39**
One agent versus multiple agents 40
Risks of parallel agents 40
How to coordinate subagents 40
Parallel Agent Assignment Template 41
Part 10: Practice Standards for Code Orchestrators **41**
Part 11: Software Project Design **44**
Prompt differences by project stage 48
Part 12: Game Design and Game Development Orchestration **49**
How to prompt Codex for game work 59
Part 13: Design, UI, and UX Language **60**
How a non-designer can give Codex usable UI direction 66
Part 14: Documentation and Repo Memory **66**
Repo memory rules 70
Part 15: Failure Modes and How to Correct Them **71**
Part 16: The Orchestrator’s Checklist Library **75**
Part 17: Prompt Library **79**
Part 18: Final Field Manual **81**
10 core principles 81
10 dangerous anti-patterns 81
10 phrases every code orchestrator should use 82
10 phrases that cause chaos 82
One-page Codex task template 82
One-page review checklist 82
One-page QA checklist 83
One-page game-development orchestration checklist 83
Appendix A: Official Codex capability map versus practice standards **83**
Appendix B: Footnotes and official source links **85**

Audience: an intelligent non-programmer who wants to direct AI coding agents without becoming the kind of person who says "it works on my machine" as if that is a legal defense.

Core thesis: the future skill is not merely "knowing how to code." It is knowing how to specify, decompose, review, test, constrain, redirect, and validate code-producing agents. A code orchestrator does not need to personally implement every function, but must understand enough about systems, failure modes, testing, design intent, and technical language to prevent the agent from building nonsense very confidently.

Status note: Codex-specific statements in this report were checked against official OpenAI documentation and OpenAI engineering material on **May 6, 2026**. Product details, model names, flags, and UI labels can change. The standards in this manual are deliberately more stable than any one product screen.

Source boundary and labels

This report uses three labels:

[Official Codex] means a capability, term, or workflow is described in official OpenAI Codex documentation or OpenAI engineering material. For example, Codex CLI is documented as a local terminal coding agent that can read, change, and run code in the selected directory; Codex app is documented as a desktop experience for parallel threads with worktrees, automations, Git functionality, browser previews, skills, plugins, and review surfaces; Codex web/cloud is documented as a cloud coding agent that can work in background tasks using OpenAI-managed cloud environments.[1][2][3]

[General engineering] means ordinary software engineering, testing, Git, architecture, project management, or game-development practice. Codex can help execute it, but the concept is not a Codex feature. A unit test, a pull request, a vertical slice, a race condition, or a rollback point is not "AI terminology." It is engineering terminology. The robots did not invent branches; they merely found a way to make humans ignore them faster.

[Emerging orchestration pattern] means a useful way to coordinate coding agents that is not necessarily an official Codex product promise. Examples include assigning multiple agents to non-overlapping subsystems, maintaining repo memory through handoff notes, or treating screenshots/logs/diffs as evidence gates. These patterns are practical, but they should be treated as operating discipline, not magic.

Official Codex documentation emphasizes several ideas this manual treats as non-negotiable: clear goals, context, constraints, a definition of done, verification commands, sandboxing, permissions, durable guidance via AGENTS.md, external context via MCP, reusable workflows via skills, and isolation via worktrees or cloud environments.[4][5][6][7][8][9][10]

[1] OpenAI Codex CLI docs: https://developers.openai.com/codex/cli

[2] OpenAI Codex app docs: https://developers.openai.com/codex/app

[3] OpenAI Codex web/cloud docs: https://developers.openai.com/codex/cloud

[4] OpenAI Codex best practices: https://developers.openai.com/codex/learn/best-practices

[5] OpenAI Codex prompting docs: https://developers.openai.com/codex/prompting

[6] OpenAI Codex sandbox docs: https://developers.openai.com/codex/concepts/sandboxing

[7] OpenAI Codex AGENTS.md docs: https://developers.openai.com/codex/guides/agents-md

[8] OpenAI Codex MCP docs: https://developers.openai.com/codex/mcp

[9] OpenAI Codex skills docs: https://developers.openai.com/codex/skills

[10] OpenAI Codex app worktrees: https://developers.openai.com/codex/app/worktrees

Part 1: What Code Orchestration Is

Code orchestration is the practice of directing AI coding agents through goals, constraints, context, implementation passes, validation, review, and iteration. It is not "asking nicely." It is not typing "make game better" into a text box and hoping the silicon butler grows taste.

A code orchestrator treats Codex as an agentic development partner: powerful, fast, sometimes startlingly competent, and still capable of confidently solving the wrong problem. The orchestrator does not need to write every function personally. The orchestrator must understand enough to say: this subsystem owns state, this test proves the behavior, this diff is too broad, this bug report is incomplete, this refactor changed behavior, this acceptance criterion is not met, and this task must stop before it mutates the entire repo into modernist rubble.

[Official Codex] Codex works through prompts and threads. In a turn, it can call tools such as file reads, file edits, and command execution, then continue until it finishes or you cancel it. Codex docs explicitly recommend giving context, reproduction steps, validation steps, and smaller focused tasks for better outcomes.[11]

Roles: who does what

The following table is not about job titles as corporate identity badges. It is about functions. One human can play several roles; one Codex thread can assist several roles; none of this abolishes responsibility, despite humanity's heroic efforts.

Role	Literal meaning	In software practice	What the orchestrator borrows	What Codex can do	What Codex should not be trusted to do alone
Programmer	A person who writes program instructions.	Implements code, fixes defects, reads APIs, writes tests.	Enough vocabulary to ask for precise code changes and understand diffs.	Write or edit functions, scripts, tests, docs.	Decide product intent without constraints.
Software engineer	A programmer responsible for reliable systems, tradeoffs, maintainability, and delivery.	Designs systems that survive change, users, errors, and teammates.	Systems thinking: architecture, validation, rollback, ownership.	Suggest architecture, refactor, test, explain tradeoffs.	Guarantee correctness without evidence.
Technical lead	A person guiding technical direction across contributors.	Sets standards, decomposes work, reviews risk, coordinates implementation.	The core operating mode: define passes, acceptance criteria, review gates.	Act as reviewer, implementer, planner, triage assistant.	Be the sole authority on integration quality.
Product owner	A person responsible for what should be built and why.	Defines user value, priorities, requirements, scope, and release criteria.	Clear goals, non-goals, priority calls.	Convert goals into task lists and acceptance criteria.	Invent business priorities from vibes.
QA tester	A person who checks whether software behaves correctly.	Finds defects, writes repro steps, validates fixes, explores edge cases.	Evidence discipline: tests, logs, screenshots, repros.	Run tests, inspect logs, create defect reports, do browser or app checks where tools permit.	Declare "works" without running anything.

[11] OpenAI Codex prompting docs: https://developers.openai.com/codex/prompting

Role	Literal meaning	In software practice	What the orchestrator borrows	What Codex can do	What Codex should not be trusted to do alone
Game designer	A person who defines player experience and rules.	Specifies mechanics, loops, difficulty, feedback, balance, feel.	Mechanic-first thinking before visual polish.	Implement prototypes, simulate systems, add debug overlays, iterate on controls.	Know whether a game is fun unless playtested.
Prompt writer	A person crafting model instructions.	Frames tasks for AI outputs.	Useful, but insufficient alone. Prompting is only one part of orchestration.	Respond to clear prompts, ask clarifying questions, plan.	Replace acceptance criteria, testing, and Git discipline.
Code orchestrator	A person directing code-producing agents toward validated software changes.	Combines product intent, technical vocabulary, QA discipline, and version-control hygiene.	The target role of this manual.	Partner across planning, implementation, testing, review, documentation.	Own the final decision.
Agent operator	A person running one or more coding agents safely.	Manages permissions, branches, worktrees, tools, schedules, handoffs.	Operational control: sandbox, tools, contexts, stop conditions.	Execute bounded passes, parallelize subagents, automate recurring checks.	Run unattended broad changes with full access because "speed."

Why orchestration is not vibe coding

"Vibe coding" is what happens when the user supplies emotional atmosphere instead of technical direction: "make it cleaner," "add fun," "fix the bugs," "modernize the app," "ship it," and other phrases that should come with a warning label. It can produce demos. It can also produce a repo where the homepage is gorgeous, the save button does nothing, the tests are decorative, and the README is fan fiction.

Disciplined agent direction is different. It gives Codex a bounded problem, the files or subsystem involved, constraints, acceptance criteria, validation commands, evidence requirements, and stop conditions. It separates exploration from implementation. It reviews diffs before accepting changes. It demands logs, screenshots, test output, or build results instead of taking "this should work" as a sacred prophecy.

A competent orchestrator does not ask Codex to "improve the project." A competent orchestrator says: inspect `src/rendering/`, explain the current render pipeline, identify where frame-time spikes occur, propose a plan touching no more than three files unless necessary, implement only after approval, run `npm test` and `npm run build`, capture before/after profiler output, and report untested risks.

The Dangerous Beginner Fantasy: "The Agent Will Just Figure It Out"

The fantasy is seductive: Codex can read files, run commands, edit code, explain architecture, spawn subagents when explicitly asked, use MCP-connected tools, and work in parallel through worktrees or cloud tasks depending on surface and configuration.[12][13][14][15][16] Therefore, surely it will infer the user's true intent, select the correct tradeoffs, avoid every regression, and gracefully stop when ambiguity becomes dangerous.

[12] OpenAI Codex CLI docs: https://developers.openai.com/codex/cli

[13] OpenAI Codex subagents docs: https://developers.openai.com/codex/subagents

[14] OpenAI Codex MCP docs: https://developers.openai.com/codex/mcp

[15] OpenAI Codex app worktrees: https://developers.openai.com/codex/app/worktrees

[16] OpenAI Codex web/cloud docs: https://developers.openai.com/codex/cloud

No. That is not how serious work is done. Agents are pattern-hungry executors with tool access. They can infer, but inference is not authority. They can test, but only if the environment and commands exist. They can plan, but plans are not facts. They can create plausible documentation, including plausible lies. They can parallelize, which is also a delightful way to create four incompatible versions of reality.

Agents need the following because software is not a poem. It has state, dependencies, users, build systems, and consequences:

1. **Clear goals** so the agent knows what outcome matters.
2. **Bounded scope** so it does not wander into unrelated refactors.
3. **Existing context** so it understands architecture, conventions, and prior decisions.
4. **Acceptance criteria** so completion can be checked.
5. **Testable outcomes** so "done" means observed behavior, not eloquence.
6. **File/project awareness** so the right subsystem changes.
7. **Validation instructions** so Codex knows which build, tests, lint, screenshots, logs, or manual checks matter.
8. **Review checkpoints** so risky plans pause before broad edits.
9. **Permission limits** so the agent works inside a sandbox, approval policy, branch, worktree, or cloud environment appropriate to risk.[17][18]
10. **Clear stop conditions** so it does not keep polishing, expanding, or inventing tasks.

Bad orchestration says: "Make this app production quality." Good orchestration says: "First inspect the repo and produce a risk-ranked plan. Do not edit files. Focus on authentication, persistence, test coverage, and deployment configuration. For each risk, list evidence, affected files, and the smallest next pass."

Part 2: The Mental Model of a Codebase

A codebase is the organized body of source code, configuration, assets, tests, scripts, documentation, and project metadata that together define a piece of software. It is not just "the code." It is the recipe, the ingredients, the kitchen layout, the appliance manuals, the dirty dishes, and the sticky note saying "never run this migration twice."

When directing Codex, you must think in layers: files live in directories; files form modules and packages; packages depend on libraries and tools; build systems transform source into runnable artifacts; runtimes execute those artifacts; frameworks and engines supply structure; state and control flow determine behavior. In games, the game loop, event loop, rendering loop, input, collision, and state updates decide what the player actually experiences.

Term: Repository

Plain meaning: A storage place.

Programmer meaning: A version-controlled collection of files, usually managed by Git, containing history, branches, commits, and project content.

[17] OpenAI Codex sandbox docs: https://developers.openai.com/codex/concepts/sandboxing

[18] OpenAI Codex agent approvals and security: https://developers.openai.com/codex/agent-approvals-security

Why Codex cares: Codex needs the repo root to discover files, AGENTS.md guidance, tests, config, and Git state.

Beginner misunderstanding: They think a repo is just a folder. A repo also has history and branches.

Bad prompt: Fix the repo.

Good prompt: Inspect this repository root. Summarize the architecture, test commands, active branch, dirty files, and any AGENTS.md instructions before editing.

Bad orchestration: Starts Codex in a random subfolder and asks for global changes.

Good orchestration: Starts at repo root, verifies Git status, and defines scope.

Term: Project

Plain meaning: A planned work unit.

Programmer meaning: A software product or component with its own goals, build steps, dependencies, and deliverables.

Why Codex cares: Codex app projects correspond to selected project folders; boundaries affect sandbox and context.

Beginner misunderstanding: They assume one repo equals one project. Monorepos can contain many projects.

Bad prompt: Work on my project.

Good prompt: For this project under`apps/game-client`, inspect only that app, its package config, and local README. Do not modify sibling packages.

Bad orchestration: Lets the agent change unrelated packages.

Good orchestration: Defines the project boundary explicitly.

Term: Directory

Plain meaning: A container for files.

Programmer meaning: A folder organizing code by feature, layer, package, or asset type.

Why Codex cares: Directory boundaries help Codex find subsystems and avoid irrelevant edits.

Beginner misunderstanding: They treat folders as arbitrary clutter. Folder layout usually encodes architecture.

Bad prompt: Look everywhere and fix it.

Good prompt: Search first in`src/ui/settings/`and`src/state/preferences/`; do not edit outside those directories unless you explain why.

Bad orchestration: Asks for broad search and accepts broad changes.

Good orchestration: Names likely directories and requires justification for leaving them.

Term: File

Plain meaning: A named unit of stored content.

Programmer meaning: A source file, config file, test file, asset, or doc that can be read, edited, diffed, tested, and committed.

Why Codex cares: Codex changes files; file-level review is where you catch scope creep.

Beginner misunderstanding: They think "small change" means few words in the prompt. It means few controlled file edits.

Bad prompt: Change whatever files you need.

Good prompt: Fix the bug with minimal changes. Report every edited file and why it was necessary.

Bad orchestration: Does not inspect modified files.

Good orchestration: Reviews the diff file by file.

Term: Module

Plain meaning: A separable piece.

Programmer meaning: A source file or logical unit that exports functions/classes and imports dependencies.

Why Codex cares: Codex must respect module boundaries to avoid tangled changes.

Beginner misunderstanding: They confuse modules with random files. Modules often expose public interfaces.

Bad prompt: Rewrite the module cleaner.

Good prompt: Explain the public API ofinputController.ts, then change only its internal implementation unless tests show callers must change.

Bad orchestration: Breaks callers while "cleaning."

Good orchestration: Separates interface from implementation.

Term: Package

Plain meaning: A wrapped bundle.

Programmer meaning: A distributable or installable unit with metadata, dependencies, scripts, and versioning, such as an npm, Python, Rust, or NuGet package.

Why Codex cares: Codex may need to run package scripts and avoid adding dependencies casually.

Beginner misunderstanding: They treat adding a package as harmless. It affects build, security, size, licensing, and future maintenance.

Bad prompt: Install whatever library helps.

Good prompt: Before adding any production dependency, explain why existing packages are insufficient and ask for approval.

Bad orchestration: Allows dependency sprawl.

Good orchestration: Sets dependency approval rules.

Term: Dependency

Plain meaning: Something relied on.

Programmer meaning: External or internal code required for build/runtime/tests.

Why Codex cares: Codex must know dependencies to install, import, update, audit, or avoid.

Beginner misunderstanding: They think dependencies are free. They create lockfile changes, vulnerabilities, compatibility constraints, and licensing issues.

Bad prompt: Use a library for this.

Good prompt: Prefer existing dependencies. If a new dependency is needed, propose options with size, license, maintenance risk, and affected lockfiles before installing.

Bad orchestration: Lets Codex add packages to avoid thinking.

Good orchestration: Controls dependency changes as design decisions.

Term: Build system

Plain meaning: A system for making things.

Programmer meaning: The tools and scripts that transform source code into runnable or deployable output.

Why Codex cares: Codex must run the right build to prove the code compiles and assets bundle.

Beginner misunderstanding: They think saving a file means software is built. The build can still fail.

Bad prompt: Make it build.

Good prompt: Run the smallest relevant build command listed in package scripts. If it fails, report the exact command, error, and suspected cause before patching.

Bad orchestration: Accepts "should compile."

Good orchestration: Requires build output as evidence.

Term: Runtime

Plain meaning: The environment where something runs.

Programmer meaning: The language VM, browser, OS process, engine, container, or platform executing code.

Why Codex cares: Codex must know runtime constraints, available APIs, file paths, permissions, and environment variables.

Beginner misunderstanding: They confuse build-time and run-time errors. A thing can build and then crash instantly.

Bad prompt: Fix runtime errors.

Good prompt: Launch the app in the documented runtime, reproduce the crash, capture logs, and fix only the runtime error path.

Bad orchestration: Only runs type checks.

Good orchestration: Runs the actual app when needed.

Term: Framework

Plain meaning: A supporting structure.

Programmer meaning: A library or platform that imposes architecture and conventions, like React, Next.js, Django, Rails, Unity, Godot, Phaser, or Electron.

Why Codex cares: Codex must follow framework conventions instead of inventing architecture.

Beginner misunderstanding: They think frameworks are interchangeable decorations. They define lifecycle, routing, state, build, testing, and deployment patterns.

Bad prompt: Use whatever pattern is clean.

Good prompt: Follow this framework's existing routing/state conventions. Do not introduce a new architectural pattern without explaining the tradeoff.

Bad orchestration: Lets Codex mix incompatible patterns.

Good orchestration: Asks Codex to inspect existing conventions first.

Term: Engine

Plain meaning: A machine that drives work.

Programmer meaning: A runtime framework, especially in games, that handles rendering, physics, scenes, assets, input, and loops.

Why Codex cares: For game work, Codex must align mechanics with engine lifecycle and entity/scene systems.

Beginner misunderstanding: They think game engines are just graphics. Engines define how time, input, physics, and rendering coordinate.

Bad prompt: Make a game engine thing.

Good prompt: In the existing Phaser scene, add the mechanic using the scene update loop and current entity model; do not create a parallel engine abstraction.

Bad orchestration: Creates a second mini-engine inside the engine.

Good orchestration: Uses engine primitives correctly.

Term: API

Plain meaning: A way things talk.

Programmer meaning: An Application Programming Interface: the contract exposed by functions, modules, services, libraries, or web endpoints.

Why Codex cares: Codex must preserve or intentionally change API contracts, otherwise callers break.

Beginner misunderstanding: They think API only means web API. It can be any exposed interface.

Bad prompt: Change the API if needed.

Good prompt: Do not change public API shapes. If unavoidable, propose a migration plan and update tests/docs.

Bad orchestration: Lets Codex silently alter contracts.

Good orchestration: Treats API changes as explicit decisions.

Term: SDK

Plain meaning: A kit for development.

Programmer meaning: A Software Development Kit: tools, libraries, docs, and examples for building against a platform.

Why Codex cares: Codex may use SDK docs, versions, and examples; stale SDK assumptions cause hallucinated APIs.

Beginner misunderstanding: They think SDK examples are universal. Version matters.

Bad prompt: Use the SDK.

Good prompt: Check the installed SDK version and existing usage before adding new SDK calls. Do not invent methods not present in docs/types.

Bad orchestration: Accepts nonexistent method calls.

Good orchestration: Verifies against installed version or official docs.

Term: Library

Plain meaning: A collection to borrow from.

Programmer meaning: Reusable code called by your app without dictating the whole app structure.

Why Codex cares: Codex should use libraries intentionally and keep imports consistent.

Beginner misunderstanding: They confuse libraries with frameworks. A library helps; a framework governs.

Bad prompt: Use a helper library.

Good prompt: Use existing utility libraries in this repo. If none fit, implement a small local helper before proposing a dependency.

Bad orchestration: Adds large libraries for tiny problems.

Good orchestration: Fits tool size to problem size.

Term: Asset

Plain meaning: A useful resource.

Programmer meaning: Images, audio, fonts, models, tilemaps, shaders, animations, icons, or data files used by the app/game.

Why Codex cares: Codex may generate, move, reference, or delete assets. Asset paths and formats matter.

Beginner misunderstanding: They think assets are cosmetic. Broken assets can crash loaders or ruin gameplay/UI.

Bad prompt: Make assets better.

Good prompt: Add placeholder sprite assets underassets/sprites/, update manifests, and verify they load in the game scene without console errors.

Bad orchestration: Polishes assets while gameplay is broken.

Good orchestration: Treats assets as part of the runtime system.

Term: Configuration

Plain meaning: Settings.

Programmer meaning: Files or environment values that control behavior without changing source code.

Why Codex cares: Codex often needs config for build, lint, tests, env, routing, tools, and deployment.

Beginner misunderstanding: They think config is harmless. Bad config can disable tests, change prod behavior, or leak secrets.

Bad prompt: Update config.

Good prompt: Explain each config change, its environment impact, and how you validated it. Do not disable warnings/tests to make the build pass.

Bad orchestration: Lets Codex silence tools.

Good orchestration: Requires reason and validation for config edits.

Term: Environment

Plain meaning: Surroundings.

Programmer meaning: The machine/container/runtime variables/tools/secrets/network access available while software or Codex runs.

Why Codex cares: Codex cloud and local runs differ; setup scripts, network access, secrets, and tools vary.

Beginner misunderstanding: They assume the agent's environment equals their laptop. It often does not.

Bad prompt: Run it somewhere.

Good prompt: Report OS/runtime versions, package manager, env assumptions, and whether validation ran locally, in a worktree, or in Codex cloud.

Bad orchestration: Ignores environment mismatch.

Good orchestration: Makes environment explicit.

Term: Script

Plain meaning: Written instructions.

Programmer meaning: A command or program that automates setup, build, test, migration, generation, or deployment.

Why Codex cares: Codex can run scripts and should use project scripts rather than ad hoc commands.

Beginner misunderstanding: They treat scripts as opaque magic. Scripts are often the project's official workflow.

Bad prompt: Run tests.

Good prompt: Inspect package scripts, identify the smallest relevant validation command, run it, and report exact output.

Bad orchestration: Runs arbitrary commands and misses the standard suite.

Good orchestration: Uses documented scripts first.

Term: Toolchain

Plain meaning: A chain of tools.

Programmer meaning: The connected set of compiler, bundler, formatter, linter, test runner, package manager, framework CLI, and deployment tools.

Why Codex cares: Codex needs the toolchain to validate changes in the same way the project does.

Beginner misunderstanding: They think one tool is enough. Real validation is usually a chain.

Bad prompt: Make the toolchain pass.

Good prompt: Run formatting, typecheck, unit tests, and build in that order if those scripts exist; stop at first failure and report it.

Bad orchestration: Only runs the final build.

Good orchestration: Uses staged validation.

Term: Entry point

Plain meaning: A place where execution begins.

Programmer meaning: The file/function/route/scene/process where code starts running.

Why Codex cares: Codex must find entry points to trace behavior and wire features.

Beginner misunderstanding: They assume any file can affect runtime. Some files are never reached.

Bad prompt: Find where it starts.

Good prompt: Identify the app entry point, route entry, and game scene entry before changing initialization logic.

Bad orchestration: Patches unused code.

Good orchestration: Traces from entry point to behavior.

Term: State

Plain meaning: Condition at a moment.

Programmer meaning: Stored data representing current app/game status: user input, UI mode, session, entities, score, auth, cache, etc.

Why Codex cares: Most bugs are state bugs wearing a little hat. Codex must understand who owns and mutates state.

Beginner misunderstanding: They think state is "variables." State is system memory over time.

Bad prompt: Fix state.

Good prompt: Trace where`playerState.health`is initialized, updated, rendered, and reset. Do not add duplicate state.

Bad orchestration: Adds parallel state variables.

Good orchestration: Finds state ownership before changing it.

Term: Data model

Plain meaning: The shape of data.

Programmer meaning: The schema, classes, types, database tables, JSON structures, and relationships representing domain entities.

Why Codex cares: Codex must keep data shape consistent across storage, validation, UI, and APIs.

Beginner misunderstanding: They think changing a field name is local. It can affect migrations, clients, tests, docs, and saved data.

Bad prompt: Rename this data.

Good prompt: Before changing the`PlayerProfile`schema, list all readers/writers, required migrations, backward compatibility risks, and tests to update.

Bad orchestration: Changes model and leaves clients broken.

Good orchestration: Plans model changes across layers.

Term: Control flow

Plain meaning: The path instructions take.

Programmer meaning: The order of operations, branches, loops, callbacks, async promises, and error paths that determine behavior.

Why Codex cares: Codex must alter the right path, not just add code that never runs.

Beginner misunderstanding: They read code top-to-bottom even when runtime is event-driven or async.

Bad prompt: Add logic here.

Good prompt: Trace the control flow from button click to save request and back to UI state before patching.

Bad orchestration: Adds logic to a nearby file without proving it executes.

Good orchestration: Traces actual execution path.

Term: Event loop

Plain meaning: A loop handling events.

Programmer meaning: The runtime mechanism that schedules callbacks, input, timers, I/O, promises, UI events, and rendering work.

Why Codex cares: Codex must avoid blocking the event loop and understand async timing.

Beginner misunderstanding: They think async means "later, somehow." It affects race conditions and UI freezes.

Bad prompt: Make it async.

Good prompt: Do not block the event loop. If adding async work, handle cancellation, loading state, and stale responses.

Bad orchestration: Adds slow synchronous work to UI thread.

Good orchestration: Designs around scheduling and responsiveness.

Term: Rendering loop

Plain meaning: A repeated drawing cycle.

Programmer meaning: The cycle that prepares and draws frames to the screen, often 60+ times per second.

Why Codex cares: Game/UI performance depends on doing only appropriate work per frame.

Beginner misunderstanding: They think drawing code runs once. It often runs every frame.

Bad prompt: Fix rendering.

Good prompt: Inspect what runs per frame. Move expensive one-time setup out of the render loop and validate frame-time improvement.

Bad orchestration: Adds allocations in every frame.

Good orchestration: Separates setup, update, and draw.

Term: Game loop

Plain meaning: A repeated gameplay cycle.

Programmer meaning: The loop that processes input, updates simulation, handles collisions/state, and renders frames.

Why Codex cares: Codex must implement mechanics at the right point in the loop.

Beginner misunderstanding: They think "game loop" means "the game concept." It is a technical update cycle.

Bad prompt: Add gameplay.

Good prompt: Add the boost mechanic by updating input handling, simulation state, collision consequences, and HUD rendering in the existing game loop.

Bad orchestration: Only adds a sprite or button.

Good orchestration: Connects mechanic to simulation and feedback.

Part 3: How Codex Should Be Directed

Directing Codex starts by naming the kind of work you want. "Help with code" is not a task. It is a cry for structure. Codex tasks differ by the action expected, the amount of freedom allowed, and the validation required.

Task verb	What it means	Orchestrator instruction
Explain	No edits. Codex should translate code, architecture, or behavior into a clear model. Use when you need understanding before decisions.	Ask for responsibilities, data flow, file map, gotchas, and uncertainty. Stop before implementation.
Inspect	No or minimal edits. Codex should read files, run safe commands, check Git status, inspect configs, and report findings.	Use before feature work, refactors, migrations, performance work, or unfamiliar repos.
Plan	No edits unless explicitly allowed. Codex should propose steps, risks, affected files, tests, and rollback.	Use for multi-step or risky work; approve or revise before implementation.
Implement	Codex changes files to add or fix behavior.	Use only after scope and acceptance criteria are clear.
Refactor	Codex changes structure without intended behavior change.	Require before/after tests and a diff summary. Behavior preservation is the point.
Debug	Codex investigates failure from symptoms to root cause.	Require reproduction, logs, hypotheses, minimal patch, and regression protection.
Test	Codex writes or runs tests.	Specify behavior to cover and require the suite to run. Beware test theater.
Review	Codex examines diffs for defects, regressions, missing tests, security, and maintainability.	Useful before commit, PR, or merge. In GitHub, Codex code review can be requested with @codex review when configured.[19]
Document	Codex updates docs to match implementation.	Docs must reflect actual code and validation, not optimistic fiction.
Optimize	Codex improves performance, memory, latency, or throughput.	Require measurement before and after. No profiler, no victory parade.
Compare alternatives	Codex evaluates options and tradeoffs.	Demand criteria: complexity, risk, compatibility, testability, maintenance.
Create a migration plan	Codex maps old system to new system in safe checkpoints.	Require dependencies, rollback, compatibility, data handling, validation gates.
Create a minimal reproduction	Codex builds the smallest case demonstrating a bug.	Use to isolate defects before broad fixes.
Generate assets or prototypes	Codex creates draft assets or playable proof-of-concepts.	Separate prototype permission from production permission.
Perform QA passes	Codex exercises flows, captures evidence, logs defects, and checks regressions.	Use browser/app tools where configured; require screenshots, logs, repro steps, and known gaps.

[19] OpenAI Codex GitHub code review docs: https://developers.openai.com/codex/integrations/github

Taxonomy of Codex task types

1. Reconnaissance tasks

Purpose: Inspect before touching. Use when you inherit a repo, subsystem, build failure, or unfamiliar architecture.

Pasteable prompt: Inspect this repo and explain the architecture before changing anything. Include entry points, important directories, test/build commands, AGENTS.md rules, current Git status, and top risks.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

2. Planning tasks

Purpose: Ask for a plan with risk, affected files, validation, and rollback before work starts.

Pasteable prompt: Create an implementation plan with risks, affected files, validation steps, and rollback strategy. Do not edit files.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

3. Surgical implementation tasks

Purpose: Make the smallest safe code change to address one bug or defect.

Pasteable prompt: Fix this one bug with minimal changes and do not refactor unrelated code. Add a regression test if feasible and report exact validation commands.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

4. Feature implementation tasks

Purpose: Add behavior according to acceptance criteria.

Pasteable prompt: Add mechanic X according to acceptance criteria Y. Keep visuals placeholder-level until simulation is correct.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

5. Refactor tasks

Purpose: Change internal structure while preserving behavior.

Pasteable prompt: Restructure this subsystem without changing behavior. First identify current public interfaces and tests, then refactor in small commits.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

6. Testing tasks

Purpose: Add tests or improve coverage for specified behavior.

Pasteable prompt: Add unit tests for this behavior and run the relevant test suite. Report what remains untested.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

7. QA tasks

Purpose: Exercise the app/game manually or through tools and produce defect reports.

Pasteable prompt: Launch the app/game, perform manual interaction, capture evidence, and log defects with repro steps.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

8. Documentation tasks

Purpose: Update docs to match reality.

Pasteable prompt: Update README and agent handoff notes to reflect the new architecture, commands, and known limitations.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

9. Cleanup tasks

Purpose: Remove dead code/assets/docs after proving they are unused.

Pasteable prompt: Remove dead code, unused assets, and stale docs after verifying they are unused. Do not delete anything referenced by build/runtime/tests.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

10. Handoff tasks

Purpose: Summarize state so the next agent or human can continue without archaeology.

Pasteable prompt: Summarize current state for the next agent pass, including completed work, unresolved issues, exact next steps, validation results, and modified files.

Good orchestration: defines the task type, scope, validation, and stop condition.

Bad orchestration: mixes inspection, implementation, cleanup, docs, and QA into one vague command, then wonders why the diff looks like a raccoon edited it.

Part 4: Prompting Standards for Codex

[Official Codex] OpenAI's Codex prompting guidance recommends giving Codex a goal, relevant context, constraints, and a definition of done. The docs also emphasize verification: include steps to reproduce issues, validate features, run linting, and use smaller focused tasks when possible.[20][21]

The standard below is deliberately stricter than "write clear prompts." Serious coding work needs a spec that the agent can execute and that you can review.

Reusable serious Codex prompt template

```
Objective:
Context:
Scope:
Do not:
Constraints:
Relevant files:
Expected behavior:
Acceptance criteria:
Validation:
Evidence to report:
Risk areas:
Stop conditions:
When finished:
```

This template is not bureaucracy. It is a cheap guardrail against expensive nonsense.

Fixing a UI bug

```
Objective: Fix the settings panel overflow on narrow screens.
Context: The bug appears at 390px width on `/settings`. The Save and Cancel buttons overflow the card.
Scope: Inspect `src/pages/settings`, `src/components/FormActions`, and related CSS only.
Do not: Redesign the whole page, change API behavior, or modify unrelated layout components.
Constraints: Preserve keyboard navigation and accessible labels.
Relevant files: `src/pages/settings/*`, `src/components/FormActions.*`, `src/styles/forms.css`.
Expected behavior: At 320-480px width, buttons wrap or stack cleanly without horizontal scroll.
Acceptance criteria: No horizontal overflow; form still saves; buttons remain visible and keyboard reachable.
Validation: Run `npm run lint`, `npm test -- FormActions`, and use browser/devtools or screenshot at 390px.
Evidence to report: Modified files, screenshot path or description, commands run, test results.
Risk areas: Shared form action component may affect profile and billing pages.
Stop conditions: Stop and ask before changing shared layout primitives outside listed files.
When finished: Summarize fix, validation, and any pages not checked.
```

Adding a game mechanic

```
Objective: Add a dash/boost mechanic to the lightcycle prototype.
Context: The current game has player movement, trails, collision death, and restart.
Scope: Implement simulation first, then minimal HUD feedback.
Do not: Replace the movement system, add new art, or rebalance bots.
Constraints: Dash lasts 0.35 seconds, has 2.5 second cooldown, cannot pass through trails or walls.
Relevant files: `src/game/input.ts`, `src/game/simulation.ts`, `src/game/hud.ts`, tests under `src/game/__tests__`.
Expected behavior: Press Space to dash in current direction; cooldown prevents spam; collision still works.
Acceptance criteria: Unit tests cover cooldown, collision during dash, and reset after death.
```

[20] OpenAI Codex best practices: https://developers.openai.com/codex/learn/best-practices

[21] OpenAI Codex prompting docs: https://developers.openai.com/codex/prompting

```
Validation: Run game tests and launch the prototype. Report controls and observed result.
Evidence to report: Test output, changed files, any untested edge cases.
Risk areas: Timing, collision tunneling, input repeat behavior.
Stop conditions: Stop before adding particles or visual polish.
When finished: Add a short handoff note describing the mechanic state.
```

Refactoring a renderer

```
Objective: Refactor renderer initialization to separate asset loading from per-frame drawing.
Context: Frame-time spikes occur because sprite atlases are checked during each frame.
Scope: `src/rendering/*` only unless tests reveal a caller contract issue.
Do not: Change visual output, game rules, input, or asset formats.
Constraints: Preserve existing public renderer API.
Relevant files: `src/rendering/renderer.ts`, `src/rendering/assets.ts`, `src/game/scene.ts`.
Expected behavior: Same visuals; less work inside frame loop.
Acceptance criteria: Existing render tests pass; no runtime asset errors; per-frame code no longer loads/checks atlases.
Validation: Run tests, build, and capture before/after profiler or console timing if available.
Evidence to report: Diff summary, public API confirmation, validation output.
Risk areas: Asset lifecycle, async loading, scene transitions.
Stop conditions: Stop before changing entity model or scene ownership.
When finished: Document new renderer responsibilities in architecture notes.
```

Improving performance

```
Objective: Reduce initial dashboard render time without changing behavior.
Context: Users report slow page load; suspected expensive data transforms in render.
Scope: Measure, identify top bottleneck, make one safe optimization.
Do not: Remove features, skip data validation, or change API responses.
Constraints: Use existing profiling/test tools.
Relevant files: `src/dashboard/*`, `src/data/transform.ts`.
Expected behavior: Same dashboard content, lower render or transform time.
Acceptance criteria: Before/after measurement shows improvement; tests pass.
Validation: Run profiler or timing script, `npm test -- dashboard`, and build.
Evidence to report: Baseline metric, changed metric, commands, remaining bottlenecks.
Risk areas: Memoization correctness, stale cache, data edge cases.
Stop conditions: Stop if no reliable measurement can be produced.
When finished: Report whether the improvement is measured or only inferred.
```

Writing tests

```
Objective: Add regression tests for failed save persistence.
Context: Settings toggle shows Saved but resets after refresh.
Scope: Tests only unless you discover the current code is untestable; then stop and propose minimal seam.
Do not: Change production behavior in this pass.
Constraints: Follow existing test style and fixtures.
Relevant files: `src/settings/*`, `src/settings/__tests__/*`.
Expected behavior: Test fails on current bug and passes after a real fix.
Acceptance criteria: Test covers save request, persisted state, and refresh/readback.
Validation: Run the smallest relevant test command.
Evidence to report: Test file, command, result, whether test would fail before fix.
Risk areas: Mocked persistence may hide real storage bug.
Stop conditions: Stop before rewriting test framework.
When finished: Explain what behavior is still untested.
```

Debugging a failing build

```
Objective: Diagnose and fix the current build failure.
Context: `npm run build` fails after the last Codex pass.
Scope: Start by reproducing the failure and identifying the first real error.
Do not: Delete files, disable type checking, or suppress errors without explaining root cause.
Constraints: Minimal fix; no broad refactor.
Relevant files: Determine from build output.
Expected behavior: Build completes successfully.
Acceptance criteria: `npm run build` passes and no unrelated diffs are introduced.
Validation: Run `npm run build`; if tests are affected, run relevant tests.
Evidence to report: Original error, root cause, changed files, final command output.
```

```
Risk areas: Type definitions, generated files, dependency version mismatch.
Stop conditions: Stop if fix requires dependency upgrade or public API change.
When finished: Summarize root cause and prevention.
```

Cleaning up a messy repo

```
Objective: Identify safe cleanup candidates; do not delete yet.
Context: Repo has stale docs, unused assets, and dead prototype code.
Scope: Reconnaissance pass only.
Do not: Delete files in this pass.
Constraints: Evidence required for every candidate.
Relevant files: Entire repo, but prioritize `docs/`, `assets/`, `prototype/`.
Expected behavior: Produce a cleanup plan with risk levels.
Acceptance criteria: Each candidate lists evidence: no imports, no references, no build usage, no docs dependency.
Validation: Use search, build config, tests, and asset manifests.
Evidence to report: Candidate path, evidence, risk, proposed action.
Risk areas: Dynamic imports, generated manifests, deployment files.
Stop conditions: Stop after plan.
When finished: Write `cleanup-plan.md` for review.
```

Creating a prototype

```
Objective: Build a minimal playable Tron-style lightcycle prototype.
Context: Browser game prototype; mechanics matter more than art.
Scope: One local HTML/JS or existing app scaffold; placeholder visuals only.
Do not: Add backend, account system, fancy menus, or asset pipeline.
Constraints: Must be playable with keyboard, restartable after death, and deterministic enough to test.
Relevant files: Create under `prototype/lightcycle/` unless existing structure dictates otherwise.
Expected behavior: Player moves in grid/arena, leaves trail, dies on collision, can restart.
Acceptance criteria: Playable loop, controls documented, collision works, no console errors.
Validation: Launch locally, interact manually or with browser tooling, report screenshot/console result.
Evidence to report: Files created, how to run, observed gameplay, known limitations.
Risk areas: Input timing, collision edge cases, restart state.
Stop conditions: Stop before visual polish beyond simple colors.
When finished: Add `README.md` and next-step handoff.
```

Updating docs after implementation

```
Objective: Update documentation to match the implemented save-flow architecture.
Context: Save logic moved from component state to `settingsStore`.
Scope: README architecture section and `docs/settings-flow.md`.
Do not: Claim behavior that tests did not validate.
Constraints: Short, exact, and linked to real files.
Relevant files: `README.md`, `docs/settings-flow.md`, implementation files for reference.
Expected behavior: Docs explain current save path, validation commands, and known gaps.
Acceptance criteria: No stale references to removed `useSettingsForm` ownership.
Validation: Search docs for old names; run docs lint if available.
Evidence to report: Updated sections and stale references removed.
Risk areas: Docs drifting from code.
Stop conditions: Stop before changing code.
When finished: Include a short implementation summary.
```

Preparing a handoff for another Codex agent

```
Objective: Prepare a handoff for the next agent pass.
Context: Current branch contains a partial renderer refactor and failing visual QA.
Scope: Documentation only.
Do not: Modify code.
Constraints: Be exact; no motivational fluff.
Relevant files: Changed files from `git diff --name-only`, test logs, issue notes.
Expected behavior: Next agent can continue without rereading the entire thread.
Acceptance criteria: Handoff includes completed work, open issues, changed files, validation run, failures, and next recommended pass.
Validation: Check that all modified files are listed.
Evidence to report: Handoff file path and summary.
Risk areas: Missing validation gaps; stale claims.
```

```
Stop conditions: Stop after handoff note.
When finished: Write `AGENT_HANDOFF.md`.
```

Why vague prompts cause bad outcomes

Scope explosion: The agent expands the task because the prompt did not define boundaries. Cure: list scope and non-goals.

Fake completion: The agent says it is done without running validation. Cure: require commands and evidence.

Refactor drift: The agent "cleans" unrelated code and changes behavior. Cure: separate bug fixes from refactors.

Overengineering: The agent builds generic abstractions for one local problem. Cure: ask for smallest safe design.

Test theater: The agent adds tests that assert mocks, snapshots, or implementation details but not behavior. Cure: define behavior and failure mode.

Infinite polishing loops: The agent keeps improving aesthetics while core functionality remains broken. Cure: stop conditions and priority order.

Misread architecture: The agent implements against a guessed model. Cure: reconnaissance and architecture explanation before edits.

Unverified claims: The agent infers that a fix works. Cure: logs, screenshots, tests, and repro steps.

Agent self-distraction: The agent finds adjacent problems and chases them. Cure: non-goals and "report but do not fix."

Part 5: Software Design Concepts Every Orchestrator Needs

Architecture

What it means: The high-level structure of a system: major parts, responsibilities, data flow, dependencies, and boundaries.

Why a non-programmer should care: Architecture determines where changes belong and what they can break. You cannot direct Codex well if every file looks equally relevant.

How Codex can mess it up: It may add new layers, duplicate existing patterns, or patch symptoms in the wrong subsystem.

Prompt standard: Inspect the architecture first. Identify subsystem responsibilities, public interfaces, and where this change belongs before editing.

Abstraction

What it means: A simplified interface that hides lower-level details.

Why a non-programmer should care: Good abstractions reduce complexity; bad abstractions hide reality and make debugging worse.

How Codex can mess it up: It may create a generic manager/factory/service for a one-off problem.

Prompt standard: Use the smallest abstraction that fits current repeated behavior. Do not create generic frameworks unless at least two real call sites need them.

Interface

What it means: The exposed boundary through which code is used: function signatures, API routes, class methods, component props, events.

Why a non-programmer should care: Interfaces are contracts. Changing them breaks callers.

How Codex can mess it up: It may silently alter function signatures or data shapes.

Prompt standard: Preserve public interfaces. If changing one is unavoidable, list all callers and propose a migration plan.

Implementation

What it means: The internal code that fulfills an interface.

Why a non-programmer should care: You can often improve implementation without changing behavior or callers.

How Codex can mess it up: It may confuse internal cleanup with API changes.

Prompt standard: Change implementation only. Confirm public behavior and signatures remain unchanged.

Separation of concerns

What it means: Different responsibilities belong in different places.

Why a non-programmer should care: It prevents UI code from owning persistence, rendering code from owning game rules, and tests from becoming entangled.

How Codex can mess it up: It may jam validation, rendering, storage, and business rules into one file.

Prompt standard: Keep persistence, UI state, validation, and rendering responsibilities separate according to existing architecture.

Coupling

What it means: How strongly one part depends on another.

Why a non-programmer should care: Tight coupling makes small changes break distant systems.

How Codex can mess it up: It may import internals across boundaries to make a quick fix.

Prompt standard: Avoid new cross-subsystem imports. If needed, explain coupling risk and propose a boundary-friendly alternative.

Cohesion

What it means: How well a module's parts belong together.

Why a non-programmer should care: High cohesion makes files understandable and testable.

How Codex can mess it up: It may create grab-bag utilities that collect unrelated behavior.

Prompt standard: Place new logic in the module that already owns that responsibility; do not create miscellaneous helpers.

Encapsulation

What it means: Keeping internal details private behind an interface.

Why a non-programmer should care: Encapsulation lets you change internals without breaking users.

How Codex can mess it up: It may expose internal state or mutate objects from outside.

Prompt standard: Do not expose internal mutable state. Add narrow methods or selectors if callers need access.

State management

What it means: How state is stored, updated, synchronized, and reset.

Why a non-programmer should care: Many bugs are stale, duplicated, unsynchronized, or incorrectly reset state.

How Codex can mess it up: It may add duplicate state instead of fixing the owner.

Prompt standard: Trace current state ownership before edits. Avoid duplicate state unless justified.

Data flow

What it means: How data moves through the system.

Why a non-programmer should care: Data flow reveals where validation, transformation, rendering, and persistence happen.

How Codex can mess it up: It may patch output while ignoring bad input or storage.

Prompt standard: Map data from input/source to storage to UI before changing behavior.

Dependency injection

What it means: Passing dependencies into code rather than hardcoding them.

Why a non-programmer should care: It improves testability and swaps implementations without global hacks.

How Codex can mess it up: It may overuse DI containers or, conversely, hardcode services.

Prompt standard: Use simple dependency injection for code that needs testing or environment-specific services; do not introduce a DI framework without approval.

Error handling

What it means: How failures are detected, reported, recovered from, or escalated.

Why a non-programmer should care: Good error handling prevents silent failure and impossible debugging.

How Codex can mess it up: It may catch and ignore errors, or convert real failures into green tests.

Prompt standard: Do not swallow errors. Preserve useful messages and add tests for failure paths.

Logging

What it means: Recording meaningful runtime events and errors.

Why a non-programmer should care: Logs are evidence for debugging and QA.

How Codex can mess it up: It may spam logs, log secrets, or add logs instead of fixing defects.

Prompt standard: Add targeted logs for diagnosis only, avoid PII/secrets, and remove temporary logs after validation unless they are useful telemetry.

Configuration

What it means: External settings that control behavior.

Why a non-programmer should care: Config changes can affect every environment.

How Codex can mess it up: It may disable lint/tests or change prod defaults to hide failure.

Prompt standard: Explain each config change and validate in the target environment. Never disable a check just to pass.

Performance

What it means: How efficiently software uses time, memory, CPU, GPU, network, or storage.

Why a non-programmer should care: Performance claims without measurement are folklore.

How Codex can mess it up: It may optimize the wrong bottleneck or harm correctness.

Prompt standard: Measure first, optimize one bottleneck, then measure again.

Maintainability

What it means: How easy the system is to understand, change, test, and repair.

Why a non-programmer should care: Future work depends on clear code and docs.

How Codex can mess it up: It may produce clever code that only the agent can love.

Prompt standard: Prefer readable, local, conventional code over clever abstractions.

Extensibility

What it means: How easily a system can support future changes.

Why a non-programmer should care: Useful when future variation is real; harmful when invented.

How Codex can mess it up: It may design for imaginary future requirements.

Prompt standard: Design for stated near-term extension points only; mark speculation as future work.

Technical debt

What it means: A tradeoff that makes future work harder.

Why a non-programmer should care: Some debt is acceptable; hidden debt becomes compound interest with semicolons.

How Codex can mess it up: It may add shortcuts without documenting risk.

Prompt standard: If taking a shortcut, document why, the risk, and the cleanup trigger.

Regression

What it means: A bug where previously working behavior breaks.

Why a non-programmer should care: Every change can break old behavior.

How Codex can mess it up: It may only test the new feature and miss old flows.

Prompt standard: Run relevant regression tests and add one if the bug was previously fixed or high-risk.

Edge case

What it means: A rare or boundary condition.

Why a non-programmer should care: Users live in edge cases because users are entropy in shoes.

How Codex can mess it up: It may implement the happy path only.

Prompt standard: List edge cases and test the highest-risk ones: empty, null, max, min, invalid, concurrent, interrupted.

Invariant

What it means: A condition that must always remain true.

Why a non-programmer should care: Invariants are the guardrails of state.

How Codex can mess it up: It may violate hidden invariants like "score never negative" or "one active player entity."

Prompt standard: Identify invariants for this subsystem and add assertions/tests where useful.

Contract

What it means: A promise between parts of code or systems.

Why a non-programmer should care: Contracts define inputs, outputs, side effects, errors, and timing.

How Codex can mess it up: It may break contracts silently.

Prompt standard: State the contract before changing code; preserve or explicitly migrate it.

Schema

What it means: A formal structure for data.

Why a non-programmer should care: Schemas enforce fields, types, validation, and compatibility.

How Codex can mess it up: It may change schema without migration.

Prompt standard: If schema changes, update validation, migrations, fixtures, tests, and docs.

Migration

What it means: A controlled change from one structure/version to another.

Why a non-programmer should care: Migrations touch data, users, deployments, and rollback.

How Codex can mess it up: It may write one-way migrations without rollback or compatibility.

Prompt standard: Create a migration plan with checkpoints, backward compatibility, data verification, and rollback.

Backward compatibility

What it means: New code still works with old callers/data/clients.

Why a non-programmer should care: Breaking compatibility can break real users and saved data.

How Codex can mess it up: It may assume everything updates at once.

Prompt standard: Preserve old input formats where needed and test old+new paths.

Idempotence

What it means: Running the same operation multiple times has the same effect as running it once.

Why a non-programmer should care: Important for retries, migrations, scripts, and automations.

How Codex can mess it up: It may create scripts that duplicate rows/assets/listeners on rerun.

Prompt standard: Make this script idempotent or document why it is single-use and how to guard it.

Determinism

What it means: Same input and state produce same output.

Why a non-programmer should care: Determinism makes tests reliable and game simulations debuggable.

How Codex can mess it up: It may use randomness/time without control.

Prompt standard: Inject random seeds or clocks for testable simulation behavior.

Race condition

What it means: A bug caused by timing between concurrent operations.

Why a non-programmer should care: Race bugs are hard to reproduce and easy for agents to miss.

How Codex can mess it up: It may add async work without cancellation or ordering.

Prompt standard: Check concurrent calls, stale responses, cancellation, and order of updates.

Latency

What it means: Delay before a response or action completes.

Why a non-programmer should care: Latency affects UX, games, and backend reliability.

How Codex can mess it up: It may improve throughput while worsening user-visible delay.

Prompt standard: Measure user-visible latency and avoid blocking the main/event loop.

Throughput

What it means: Amount of work completed per unit time.

Why a non-programmer should care: Important for servers, pipelines, simulations, and batch jobs.

How Codex can mess it up: It may increase throughput by losing correctness or backpressure.

Prompt standard: Optimize throughput only after defining correctness, capacity target, and failure behavior.

Part 6: Testing and Validation

"It compiles" means the compiler or build tool could assemble something. It does not mean the right thing happens, the UI works, the database is safe, the game is fun, the save button persists, the edge cases behave, or the user is not staring at a spinner until the heat death of the universe.

Testing is not one thing. It is a stack of evidence. Different evidence answers different questions.

Validation type	What it checks	When to use	Codex failure risk
Unit tests	Test a small function/module in isolation.	Use for pure logic, calculations, state transitions, validators.	May overmock and miss integration behavior.
Integration tests	Test multiple components working together.	Use for API + database, UI + store, game systems interacting.	May be slower and require setup.
End-to-end tests	Test the app as a user would across full flows.	Use for critical user journeys.	Can be flaky if environment is unstable.
Regression tests	Test that a previously broken behavior stays fixed.	Use for every bug worth remembering.	Often skipped because the agent thinks the current fix is enough.
Smoke tests	Quick checks that the app launches and core routes/flows are not obviously broken.	Use after large changes or before release.	Not enough for correctness.
Snapshot tests	Compare output/UI structure to a saved snapshot.	Useful for stable rendering or serialization.	Can become approval theater: update snapshot, pretend behavior is validated.
Property-based tests	Generate many inputs to check invariants.	Useful for parsers, transformations, game rules, calculations.	Requires clear properties/invariants.
Manual QA	Human or agent-driven interaction with the product.	Needed for UX, games, visual flows, and exploratory bugs.	Not repeatable unless written as repro steps.
Visual QA	Check rendered appearance through screenshots, browser previews, or app inspection.	Needed for layout, responsive behavior, game visuals, animation, canvas output.	Visual similarity is not functional correctness.
Performance testing	Measure speed, memory, frame time, throughput, or latency.	Needed for optimization claims.	Meaningless without baseline and controlled conditions.
Fuzz testing	Feed random/malformed inputs to find crashes.	Useful for parsers, APIs, serialization, and robust input handling.	Can produce noisy failures without triage.
Static analysis	Analyze code without running it.	Catches type, security, lint, style, and bug patterns.	Does not prove runtime behavior.
Type checking	Verify values conform to declared types.	Catches mismatched data shapes and unsafe calls.	Types can be too loose or bypassed.

Validation type	What it checks	When to use	Codex failure risk
Linting	Check style, correctness patterns, and conventions.	Maintains consistency and catches simple bugs.	Can be disabled or gamed if Codex gets desperate.
Build validation	Compile/bundle/package the app.	Proves code can become runnable output.	Not a behavior test.

How to make Codex prove work

Run tests: Ask for exact commands and pass/fail results, not "tests pass" as a vibe.

Run build: Require the actual build command and first failure if any.

Launch app: For UI/games, require local launch or explicit statement that launch was not possible.

Inspect logs: Logs reveal crashes, warnings, network failures, asset load errors, and timing issues.

Capture screenshots: Screenshots help validate visual fixes, but still pair them with behavior checks.

Compare before/after behavior: Regression work needs observed before and after, not only final state.

Write repro steps: A defect without repro steps is a ghost story.

Create defect reports: Each bug gets observed behavior, expected behavior, steps, evidence, severity, and suspected area.

Identify untested risk: A mature agent says what it did not test. Demand that.

Document validation gaps honestly: No cover-up wording. "Could not run browser due missing dependency" is better than "should work."

The Agent's Favorite Lie: "This Should Work"

"This should work" is not evidence. It is a prediction. Predictions are useful before validation and dangerous after the agent starts treating them as completion.

The orchestrator's replacement phrase is: **"Show evidence."** Evidence can be test output, build output, logs, screenshots, profiler data, a reproduced bug turning into a non-reproducing bug, a diff reviewed against acceptance criteria, or a documented reason validation could not run. A good final Codex report says: "Ran `npm test -- settings`, 12 passed; ran `npm run build`, passed; manually verified `/settings` at 390px with no horizontal overflow; did not test billing settings page, which uses the same component."

A bad final report says: "Implemented the fix. This should work." That is not a report. That is a fortune cookie with file edits.

Part 7: Debugging as a Discipline

Term: Bug

Plain meaning: An error in software behavior.

Programmer meaning: Observed behavior differs from expected behavior.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the bug.

Good prompt: Debug using this bug context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Defect

Plain meaning: A flaw in implementation, design, requirement, test, or documentation.

Programmer meaning: More formal than bug; useful in QA logs.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the defect.

Good prompt: Debug using this defect context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Crash

Plain meaning: Program stops unexpectedly.

Programmer meaning: Often produces logs, stack traces, core dumps, or browser console errors.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the crash.

Good prompt: Debug using this crash context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Exception

Plain meaning: A runtime error object or event.

Programmer meaning: Shows where execution failed, though not always root cause.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the exception.

Good prompt: Debug using this exception context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Stack trace

Plain meaning: A list of function calls leading to an error.

Programmer meaning: Follow from top error to relevant app frame, not just library noise.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the stack trace.

Good prompt: Debug using this stack trace context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Log

Plain meaning: Recorded runtime information.

Programmer meaning: Evidence for timing, state, inputs, errors, and side effects.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the log.

Good prompt: Debug using this log context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Reproduction

Plain meaning: Steps that reliably cause the issue.

Programmer meaning: Without repro, fixes are guesswork.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the reproduction.

Good prompt: Debug using this reproduction context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Minimal reproduction

Plain meaning: Smallest case that demonstrates the bug.

Programmer meaning: Removes unrelated complexity.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the minimal reproduction.

Good prompt: Debug using this minimal reproduction context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Hypothesis

Plain meaning: A testable explanation.

Programmer meaning: Debugging advances by testing hypotheses, not collecting random edits.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the hypothesis.

Good prompt: Debug using this hypothesis context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Instrumentation

Plain meaning: Temporary or permanent measurement/logging code.

Programmer meaning: Used to observe hidden state or timing.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the instrumentation.

Good prompt: Debug using this instrumentation context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Bisecting

Plain meaning: Searching history or code by halves to find where a bug appeared.

Programmer meaning: Useful when a regression exists in Git history.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the bisecting.

Good prompt: Debug using this bisecting context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Isolation

Plain meaning: Separating the failing part from the rest.

Programmer meaning: Prevents changes to everything at once.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the isolation.

Good prompt: Debug using this isolation context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Root cause

Plain meaning: The underlying reason for the failure.

Programmer meaning: Different from symptom.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the root cause.

Good prompt: Debug using this root cause context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Symptom

Plain meaning: The visible bad behavior.

Programmer meaning: Fixing only the symptom may hide the bug.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the symptom.

Good prompt: Debug using this symptom context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Fix

Plain meaning: A change that removes the root cause.

Programmer meaning: Must be validated.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the fix.

Good prompt: Debug using this fix context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Workaround

Plain meaning: A temporary way around the problem.

Programmer meaning: Useful but should be labeled as such.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the workaround.

Good prompt: Debug using this workaround context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Term: Regression

Plain meaning: A bug introduced after earlier working behavior.

Programmer meaning: Needs a test to prevent return.

Why Codex cares: Codex needs this distinction to avoid patching symptoms, misreading logs, or declaring victory too soon.

Beginner misunderstanding: Beginners often jump from symptom directly to fix without reproducing or isolating.

Bad prompt: Fix the regression.

Good prompt: Debug using this regression context. First state observed vs expected behavior, reproduce, gather evidence, form hypotheses, then make the smallest validated fix.

Debugging workflow

1. State the observed behavior. What happened, exactly? Include error text and user-visible result. Prompt: Observed behavior: <exact result>. Do not fix yet. Restate the behavior and identify missing evidence.

2. State the expected behavior. What should have happened? Prompt: Expected behavior: <specific outcome>. Compare against observed behavior and list possible discrepancy points.

3. Identify the reproduction path. Steps from clean start to failure. Prompt: Use these repro steps. Run them if possible. If not possible, explain why and create the closest automated/manual check.

4. Gather logs/screenshots. Collect evidence before changing code. Prompt: Gather relevant logs, console output, stack traces, screenshots, and Git status before editing.

5. Form hypotheses. List plausible causes ranked by likelihood and risk. Prompt: List 3 hypotheses with evidence for/against each. Choose one to test first.

6. Test one hypothesis at a time. Avoid random patch piles. Prompt: Test only the top hypothesis. Do not make a fix until you can explain why this path is likely.

7. Make minimal changes. Patch the smallest root-cause location. Prompt: Apply the smallest fix that addresses the root cause. Do not refactor unrelated code.

8. Validate. Run repro and relevant checks. Prompt: Re-run the original repro and relevant tests/build. Report commands and results.

9. Add regression protection. Write a test or check to keep the bug from returning. Prompt: Add a regression test or explain why one is not feasible and what manual QA covers it.

10. Document the fix. Leave useful history for next person/agent. Prompt: Write a short fix summary: root cause, changed files, validation, untested risks.

Part 8: Version Control, Git, and Worktrees

Git is a distributed version-control system. In plain English: it lets you save meaningful checkpoints, compare changes, create parallel lines of work, merge work back together, and recover from bad decisions. Since bad decisions are abundant and often wearing headphones, Git is not optional.

[Official Codex] Codex app worktrees use Git worktrees under the hood for Git repositories. Official Git documentation describes a worktree as a working tree attached to the same repository, allowing multiple working trees/branches to exist in parallel.[22][23] In Codex app, worktrees are useful for letting Codex run multiple independent tasks without interfering with local work; automations can run on background worktrees for Git repos.[24][25]

Term: Git

Plain meaning: A version control system.

Programmer meaning: Tracks changes, branches, commits, merges, remotes, and history.

[22] OpenAI Codex app worktrees: https://developers.openai.com/codex/app/worktrees

[23] Official Git worktree docs: https://git-scm.com/docs/git-worktree

[24] OpenAI Codex app worktrees: https://developers.openai.com/codex/app/worktrees

[25] OpenAI Codex app automations: https://developers.openai.com/codex/app/automations

Why Codex cares: Codex changes agent orchestration because every pass can become a diff, branch, checkpoint, or rollback.

Beginner misunderstanding: They use Git only after disaster. Use it before.

Bad prompt: Git it.

Good prompt: Use git deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Repository

Plain meaning: A Git-tracked project.

Programmer meaning: Contains working files plus Git history/metadata.

Why Codex cares: Codex should start at repo root or intended project folder.

Beginner misunderstanding: They ignore Git status before agent work.

Bad prompt: Repository it.

Good prompt: Use repository deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Commit

Plain meaning: A saved snapshot/change record.

Programmer meaning: A coherent change with message and metadata.

Why Codex cares: Commit after validated coherent changes so you can revert safely.

Beginner misunderstanding: They commit giant mixed agent outputs.

Bad prompt: Commit it.

Good prompt: Use commit deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Branch

Plain meaning: A line of development.

Programmer meaning: An isolated name pointing to a series of commits.

Why Codex cares: Use feature branches for agent tasks and experiments.

Beginner misunderstanding: They let experimental Codex work hit main.

Bad prompt: Branch it.

Good prompt: Use branch deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Merge

Plain meaning: Combine histories.

Programmer meaning: Bring branch changes together, possibly with conflicts.

Why Codex cares: Integration is a review event, not a ceremonial button click.

Beginner misunderstanding: They merge untested diffs.

Bad prompt: Merge it.

Good prompt: Use merge deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Rebase

Plain meaning: Replay commits onto a new base.

Programmer meaning: Rewrites local history to create a linear sequence.

Why Codex cares: Useful for cleanup, risky on public/shared branches.

Beginner misunderstanding: They rebase branches other people use.

Bad prompt: Rebase it.

Good prompt: Use rebase deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Diff

Plain meaning: The set of changes between versions.

Programmer meaning: Shows added, removed, and modified lines/files.

Why Codex cares: Review diffs before accepting Codex work.

Beginner misunderstanding: They read the final answer but not the diff.

Bad prompt: Diff it.

Good prompt: Use diff deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Pull request

Plain meaning: A review request to merge changes.

Programmer meaning: A Git hosting workflow for review, discussion, CI, and merge.

Why Codex cares: Use PRs as review surfaces; Codex can assist with PR review when configured.

Beginner misunderstanding: They treat PR as paperwork after the fact.

Bad prompt: Pull request it.

Good prompt: Use pull request deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Conflict

Plain meaning: Incompatible changes needing resolution.

Programmer meaning: Git cannot auto-merge edits to same content/logic.

Why Codex cares: Parallel agents increase conflict risk.

Beginner misunderstanding: They let two agents edit same files.

Bad prompt: Conflict it.

Good prompt: Use conflict deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Stash

Plain meaning: Temporary storage for uncommitted changes.

Programmer meaning: Parks work so you can switch context.

Why Codex cares: Useful before risky agent experiments.

Beginner misunderstanding: They lose local changes.

Bad prompt: Stash it.

Good prompt: Use stash deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Tag

Plain meaning: A named point in history.

Programmer meaning: Often marks releases.

Why Codex cares: Use for release checkpoints.

Beginner misunderstanding: They cannot identify what shipped.

Bad prompt: Tag it.

Good prompt: Use tag deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Remote

Plain meaning: A repository hosted elsewhere.

Programmer meaning: GitHub/GitLab/etc copy for collaboration.

Why Codex cares: Cloud tasks and PRs depend on pushed branches/remotes.

Beginner misunderstanding: They work locally then wonder why cloud cannot see changes.

Bad prompt: Remote it.

Good prompt: Use remote deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Main branch

Plain meaning: The primary integration branch.

Programmer meaning: Often main or master, should remain stable.

Why Codex cares: Do not let broad agent experiments happen directly on main.

Beginner misunderstanding: They make main the sandbox.

Bad prompt: Main branch it.

Good prompt: Use main branch deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Feature branch

Plain meaning: A branch for one feature/fix.

Programmer meaning: Isolates a coherent change.

Why Codex cares: Good unit for Codex implementation passes.

Beginner misunderstanding: They mix five tasks in one feature branch.

Bad prompt: Feature branch it.

Good prompt: Use feature branch deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Worktree

Plain meaning: A second checkout attached to same repo.

Programmer meaning: Allows parallel work in separate directories sharing Git metadata.

Why Codex cares: Use for parallel agents or background automations.

Beginner misunderstanding: They run parallel agents on same files in same checkout.

Bad prompt: Worktree it.

Good prompt: Use worktree deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Checkpoint

Plain meaning: A safe saved state.

Programmer meaning: Usually a commit, tag, branch, or stash.

Why Codex cares: Create before risky refactors or migrations.

Beginner misunderstanding: They rely on undo buttons and prayer.

Bad prompt: Checkpoint it.

Good prompt: Use checkpoint deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Revert

Plain meaning: Create a new commit undoing previous changes.

Programmer meaning: Safe undo preserving history.

Why Codex cares: Use when bad agent changes were committed.

Beginner misunderstanding: They manually delete random lines.

Bad prompt: Revert it.

Good prompt: Use revert deliberately: explain current status, intended action, risk, and validation before proceeding.

Term: Rollback

Plain meaning: Return system/code/deployment to previous known-good state.

Programmer meaning: Can involve Git, database, deployment, config.

Why Codex cares: Must be planned for risky changes.

Beginner misunderstanding: They only plan the forward path.

Bad prompt: Rollback it.

Good prompt: Use rollback deliberately: explain current status, intended action, risk, and validation before proceeding.

Practical Git standards for agent work

Commit after coherent validated changes. A commit should represent one understandable unit that passed relevant checks.

Do not mix unrelated work. One branch should not contain bug fix, UI redesign, dependency upgrade, and doc rewrite unless that was the actual task.

Review diffs before accepting. The final answer is not the artifact; the diff is.

Preserve rollback points. Create commits/branches/tags before risky refactors and migrations.

Require test evidence before merge. A merge without validation is a polite way of distributing uncertainty.

Use branches/worktrees for parallel agents. Parallel threads need isolated file areas or Git worktrees to avoid conflicts.

Part 9: Agentic Workflow Design

A strong Codex workflow is a series of passes. Passes keep the agent focused and make review possible. One giant prompt that asks for planning, implementation, testing, cleanup, documentation, UI polish, and release notes is not ambitious. It is a blender with a login screen.

Recon pass: Read only. Identify architecture, files, current behavior, tests, risks, Git state.

Plan pass: No edits. Propose implementation steps, affected files, validation, risks, rollback.

Implementation pass: Make scoped changes only. Keep diff coherent.

Validation pass: Run tests/build/app, reproduce behavior, capture evidence.

QA pass: Exercise flows manually or through tools. Log defects and gaps.

Cleanup pass: Remove dead/temporary code only after proving it is unused.

Documentation pass: Update docs, handoff notes, test plans, known issues.

Handoff pass: Summarize state for next agent/human.

One agent versus multiple agents

Use one agent when work is small, sequential, high-context, or touches a single subsystem. One thread preserves context and reduces coordination overhead.

Use multiple agents when work can be separated into independent domains: one agent inspects tests, another reviews security, another prototypes UI alternatives, another audits docs. [Official Codex] Codex subagents are surfaced in the app and CLI, spawn only when explicitly asked, consume additional tokens, and inherit sandbox policy; Codex handles spawning, routing, waiting, and consolidating results.[26]

Do not parallelize work just because you can. Parallelism turns unclear architecture into a committee of ghosts.

Risks of parallel agents

Conflicting edits: Detect by requiring each agent to list scope, files, assumptions, validation, and open questions before integration.

Duplicate work: Detect by requiring each agent to list scope, files, assumptions, validation, and open questions before integration.

Inconsistent architecture: Detect by requiring each agent to list scope, files, assumptions, validation, and open questions before integration.

Divergent assumptions: Detect by requiring each agent to list scope, files, assumptions, validation, and open questions before integration.

Merge conflicts: Detect by requiring each agent to list scope, files, assumptions, validation, and open questions before integration.

Test gaps: Detect by requiring each agent to list scope, files, assumptions, validation, and open questions before integration.

Infinite local optimization: Detect by requiring each agent to list scope, files, assumptions, validation, and open questions before integration.

How to coordinate subagents

Give each agent a bounded domain. Example: renderer only, persistence only, tests only.

Give each agent a non-overlapping file area when possible. File overlap is where conflicts and contradictory changes breed.

Require shared architecture notes. One source of truth beats six confident summaries.

Require handoff docs. Every agent must say what it changed, proved, and did not prove.

Use checklists. Checklists are boring. Boring prevents rollback.

Use Git branches/worktrees. Isolation is not optional for parallel implementation.

Reconcile changes deliberately. Integration is a separate pass.

[26] OpenAI Codex subagents docs: https://developers.openai.com/codex/subagents

Keep one orchestrator thread responsible for integration. Someone must own the final shape. Apparently civilization requires adults.

Parallel Agent Assignment Template

```
Parallel assignment name:
Integration owner:
Base branch/worktree:
Shared architecture notes:

Agent A domain:
Allowed files:
Forbidden files:
Task:
Acceptance criteria:
Validation:
Evidence required:
Handoff output:

Agent B domain:
Allowed files:
Forbidden files:
Task:
Acceptance criteria:
Validation:
Evidence required:
Handoff output:

Coordination rules:
- No agent may edit outside allowed files without stopping and explaining why.
- Each agent must report assumptions and untested risks.
- Integration owner reviews all diffs before merging.
- No merge without validation evidence.
```

Part 10: Practice Standards for Code Orchestrators

Rule 1: Never let an agent make broad changes without a plan.

Why it exists: Broad changes are where architecture, tests, and unrelated files get mangled.

What goes wrong without it: The agent refactors half the repo while fixing one bug.

Bad orchestration: Refactor the app to be cleaner.

Good orchestration: Create a plan for this refactor. List affected files, risks, tests, rollback. Do not edit yet.

Rule 2: Never accept unverified completion claims.

Why it exists: Completion is evidence, not narration.

What goes wrong without it: The agent says done while the app fails to launch.

Bad orchestration: Looks good, thanks.

Good orchestration: Run the validation commands and report exact results, screenshots/logs, and what you did not test.

Rule 3: Always define acceptance criteria.

Why it exists: The agent needs a target that can be checked.

What goes wrong without it: The feature becomes whatever the agent found easiest to implement.

Bad orchestration: Add better search.

Good orchestration: Search is done when queries by title/body return ranked results, empty queries show recent items, and tests cover both.

Rule 4: Always separate exploration from implementation.

Why it exists: Reading and changing at the same time makes assumptions expensive.

What goes wrong without it: The agent edits before understanding architecture.

Bad orchestration: Figure it out and fix it.

Good orchestration: First inspect and explain. Do not edit. Then propose a minimal fix plan.

Rule 5: Always preserve rollback points.

Why it exists: Risky changes need a way back.

What goes wrong without it: A bad agent pass becomes hand surgery with a chainsaw.

Bad orchestration: Go ahead on main.

Good orchestration: Create a branch/checkpoint before the refactor and report Git status before editing.

Rule 6: Always ask for affected files.

Why it exists: File list is the fastest scope audit.

What goes wrong without it: Hidden unrelated edits slip in.

Bad orchestration: What did you do?

Good orchestration: List every modified file and why it changed.

Rule 7: Always ask what was not tested.

Why it exists: Validation gaps matter.

What goes wrong without it: The untested shared component breaks another page.

Bad orchestration: Any issues?

Good orchestration: Report commands run, results, and explicitly list what was not tested and why.

Rule 8: Always require evidence.

Why it exists: Screenshots, logs, tests, builds, and diffs beat confidence.

What goes wrong without it: Agent optimism replaces QA.

Bad orchestration: Do you think it works?

Good orchestration: Provide evidence: commands, outputs, repro result, screenshot/log paths, diff summary.

Rule 9: Never let the agent clean up unrelated code without permission.

Why it exists: Cleanup is destructive when "unused" is guessed.

What goes wrong without it: Useful code/assets disappear.

Bad orchestration: Clean things while you are there.

Good orchestration: Report cleanup candidates with evidence; do not delete in this pass.

Rule 10: Never confuse a plausible explanation with validated behavior.

Why it exists: Explanations can be wrong or incomplete.

What goes wrong without it: You accept a theory as a fix.

Bad orchestration: That explanation makes sense.

Good orchestration: Now validate the explanation by reproducing the bug and showing the fixed behavior.

Rule 11: Prefer small coherent passes over giant vague prompts.

Why it exists: Small passes are reviewable and recoverable.

What goes wrong without it: Large prompts create massive noisy diffs.

Bad orchestration: Build the whole product.

Good orchestration: Implement milestone 1 only: login UI shell with validation but no backend integration.

Rule 12: Make the agent document state for future passes.

Why it exists: Repo memory prevents context loss.

What goes wrong without it: Next thread repeats mistakes.

Bad orchestration: Continue later.

Good orchestration: WriteAGENT_HANDOFF.mdwith completed work, changed files, validation, open issues, next steps.

Rule 13: Keep design intent explicit.

Why it exists: Agents optimize for instructions, not your private taste.

What goes wrong without it: The agent builds technically correct wrong product behavior.

Bad orchestration: Make it fun.

Good orchestration: The intended player experience is tense near-misses, fast restarts, and readable collisions; implement mechanics that serve that.

Rule 14: Treat screenshots, logs, tests, and diffs as evidence.

Why it exists: Different artifacts prove different things.

What goes wrong without it: Visual polish hides functional failure.

Bad orchestration: It looks right.

Good orchestration: Show screenshot plus console logs plus test/build output plus diff summary.

Rule 15: Use the repo itself as memory through docs, TODOs, checklists, and handoff notes.

Why it exists: Conversation context compresses and disappears; repo files persist.

What goes wrong without it: Agents relearn the same context badly.

Bad orchestration: Remember this.

Good orchestration: Update`docs/architecture.md`and`AGENT_HANDOFF.md`; keep notes short, exact, and current.

Part 11: Software Project Design

Turning an idea into software means converting desire into decisions. A project starts as an intention and becomes buildable only when it has goals, users, requirements, constraints, milestones, risks, and validation. "Make a cool game" is not a project plan. It is a weather report from inside someone's skull.

Term: Product goal

Plain meaning: The reason the product exists.

Programmer meaning: In projects, product goal turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow product goal when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a product goal.

Good prompt: Create a product goal for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: User story

Plain meaning: A user-centered statement of need.

Programmer meaning: In projects, user story turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow user story when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a user story.

Good prompt: Create a user story for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Requirements

Plain meaning: What must be true.

Programmer meaning: In projects, requirements turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow requirements when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a requirements.

Good prompt: Create a requirements for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Functional requirements

Plain meaning: Required behaviors.

Programmer meaning: In projects, functional requirements turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow functional requirements when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a functional requirements.

Good prompt: Create a functional requirements for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Non-functional requirements

Plain meaning: Quality constraints.

Programmer meaning: In projects, non-functional requirements turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow non-functional requirements when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a non-functional requirements.

Good prompt: Create a non-functional requirements for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Constraints

Plain meaning: Limits on solution.

Programmer meaning: In projects, constraints turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow constraints when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a constraints.

Good prompt: Create a constraints for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: MVP

Plain meaning: Minimum viable product.

Programmer meaning: In projects, mvp turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow mvp when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a mvp.

Good prompt: Create a mvp for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Prototype

Plain meaning: A rough experiment to learn.

Programmer meaning: In projects, prototype turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow prototype when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a prototype.

Good prompt: Create a prototype for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Vertical slice

Plain meaning: A small complete piece across all layers.

Programmer meaning: In projects, vertical slice turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow vertical slice when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a vertical slice.

Good prompt: Create a vertical slice for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Roadmap

Plain meaning: Ordered future plan.

Programmer meaning: In projects, roadmap turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow roadmap when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a roadmap.

Good prompt: Create a roadmap for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Milestone

Plain meaning: A major checkpoint.

Programmer meaning: In projects, milestone turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow milestone when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a milestone.

Good prompt: Create a milestone for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Backlog

Plain meaning: Prioritized task list.

Programmer meaning: In projects, backlog turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow backlog when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a backlog.

Good prompt: Create a backlog for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Task decomposition

Plain meaning: Breaking work into executable pieces.

Programmer meaning: In projects, task decomposition turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow task decomposition when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a task decomposition.

Good prompt: Create a task decomposition for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Acceptance criteria

Plain meaning: Conditions for done.

Programmer meaning: In projects, acceptance criteria turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow acceptance criteria when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a acceptance criteria.

Good prompt: Create a acceptance criteria for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Design document

Plain meaning: Explains player/product design.

Programmer meaning: In projects, design document turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow design document when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a design document.

Good prompt: Create a design document for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Technical specification

Plain meaning: Explains implementation plan.

Programmer meaning: In projects, technical specification turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow technical specification when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a technical specification.

Good prompt: Create a technical specification for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Architecture decision record

Plain meaning: Short record of important decision and rationale.

Programmer meaning: In projects, architecture decision record turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow architecture decision record when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a architecture decision record.

Good prompt: Create a architecture decision record for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Term: Risk register

Plain meaning: List of risks and mitigations.

Programmer meaning: In projects, risk register turns vague intention into a reviewable artifact.

Why Codex cares: Codex can create, update, or follow risk register when it is explicit.

Beginner misunderstanding: Beginners skip this and ask the agent to infer strategy.

Bad prompt: Make a risk register.

Good prompt: Create a risk register for this project. Include concrete examples, constraints, acceptance criteria, and validation steps.

Prompt differences by project stage

"Make a cool game": This is a vague creative brief. Use only for brainstorming. Prompt: Brainstorm three small browser game concepts. Do not implement. For each, define core loop, controls, win/loss, and MVP scope.

"Build a playable MVP": This means the minimum working product that proves core gameplay/value. Prompt: Build only the playable MVP: movement, trail collision, restart, and basic HUD. Placeholder visuals. Validate by launching locally.

"Build a vertical slice": This means one complete polished cross-section. Prompt: Create one production-representative level with final-ish controls, visuals, audio cues, HUD, and QA checklist. Do not expand content volume.

"Build production-quality software": This means reliability, tests, deployment, monitoring, security, docs, edge cases. Prompt: Create a production readiness plan first: requirements, risks, tests, deployment, monitoring, rollback, docs. Do not implement until approved.

"Polish the prototype": This means improve presentation while preserving prototype scope. Prompt: Polish visuals and feedback only after confirming core mechanics pass. Do not add new mechanics.

"Refactor the architecture": This means restructure without changing behavior. Prompt: Plan the architecture refactor, list public interfaces and tests, create checkpoints, and preserve behavior.

"Ship a release": This means final validation, packaging, release notes, rollback, and known issues. Prompt: Prepare release checklist, run validation, create release notes, tag version, and document rollback path.

Part 12: Game Design and Game Development Orchestration

Game development is where agent orchestration gets both exciting and stupidly easy to derail. A game is not a pile of visuals. It is a simulation, control scheme, feedback system, and player-learning loop wrapped in art and sound. Build the simulation first. Then make it pretty. This is not anti-art. It is anti-spending seven hours polishing one wall while the rest of the arena does not exist, because apparently even machines can discover bureaucracy.

[Official Codex] OpenAI's Codex game-development use case recommends starting browser game work with a concrete PLAN.md covering player goal, main loop, controls, win/fail states, difficulty, visual direction, stack, hosting assumptions, and milestone order; it also recommends live browser testing, reusable asset prompts, and AGENTS.md guidance.[27]

Term: Game loop

Plain meaning: Repeated cycle of input, simulation update, collision, state changes, and rendering.

Programmer meaning: Game developers use game loop to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the game loop better.

Good prompt: Implement/inspect the game loop in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

[27] OpenAI Codex create browser-based games use case: https://developers.openai.com/codex/use-cases/browser-games

Term: Core mechanic

Plain meaning: The main repeated action that creates the game: steering a lightcycle, jumping, matching tiles, aiming.

Programmer meaning: Game developers use core mechanic to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the core mechanic better.

Good prompt: Implement/inspect the core mechanic in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Secondary mechanic

Plain meaning: Supporting action layered on core play: boost, shield, hazard trigger, combo.

Programmer meaning: Game developers use secondary mechanic to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the secondary mechanic better.

Good prompt: Implement/inspect the secondary mechanic in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Systems design

Plain meaning: Rules and interacting systems: economy, spawning, health, scoring, AI, progression.

Programmer meaning: Game developers use systems design to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the systems design better.

Good prompt: Implement/inspect the systems design in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Level design

Plain meaning: Arrangement of spaces, obstacles, goals, pacing, and player choices.

Programmer meaning: Game developers use level design to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the level design better.

Good prompt: Implement/inspect the level design in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Arena design

Plain meaning: A level built for repeated contest/survival, often symmetric or readable.

Programmer meaning: Game developers use arena design to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the arena design better.

Good prompt: Implement/inspect the arena design in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Input handling

Plain meaning: How keyboard/mouse/gamepad/touch becomes game commands.

Programmer meaning: Game developers use input handling to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the input handling better.

Good prompt: Implement/inspect the input handling in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Camera

Plain meaning: The player's viewport and how it follows or frames action.

Programmer meaning: Game developers use camera to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the camera better.

Good prompt: Implement/inspect the camera in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Collision

Plain meaning: Detecting when objects overlap or hit.

Programmer meaning: Game developers use collision to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the collision better.

Good prompt: Implement/inspect the collision in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Physics

Plain meaning: Movement, forces, velocity, acceleration, collision response.

Programmer meaning: Game developers use physics to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the physics better.

Good prompt: Implement/inspect the physics in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Hitbox

Plain meaning: The area that can deal a hit.

Programmer meaning: Game developers use hitbox to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the hitbox better.

Good prompt: Implement/inspect the hitbox in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Hurtbox

Plain meaning: The area that can receive a hit.

Programmer meaning: Game developers use hurtbox to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the hurtbox better.

Good prompt: Implement/inspect the hurtbox in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Tilemap

Plain meaning: Grid-based map made from reusable tiles.

Programmer meaning: Game developers use tilemap to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the tilemap better.

Good prompt: Implement/inspect the tilemap in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Grid

Plain meaning: Discrete coordinate structure for movement, maps, or logic.

Programmer meaning: Game developers use grid to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the grid better.

Good prompt: Implement/inspect the grid in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Navmesh

Plain meaning: Navigation mesh used by AI pathfinding.

Programmer meaning: Game developers use navmesh to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the navmesh better.

Good prompt: Implement/inspect the navmesh in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Entity

Plain meaning: A game object with identity, like player, enemy, projectile, pickup.

Programmer meaning: Game developers use entity to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the entity better.

Good prompt: Implement/inspect the entity in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Component

Plain meaning: A reusable piece of entity behavior/data, like Position, Velocity, Health.

Programmer meaning: Game developers use component to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the component better.

Good prompt: Implement/inspect the component in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Scene

Plain meaning: A container for game state/rendering, such as menu scene or arena scene.

Programmer meaning: Game developers use scene to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the scene better.

Good prompt: Implement/inspect the scene in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Prefab

Plain meaning: A reusable preconfigured object/template.

Programmer meaning: Game developers use prefab to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the prefab better.

Good prompt: Implement/inspect the prefab in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Sprite

Plain meaning: A 2D image used as a visual object.

Programmer meaning: Game developers use sprite to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the sprite better.

Good prompt: Implement/inspect the sprite in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Shader

Plain meaning: A GPU program controlling rendering effects.

Programmer meaning: Game developers use shader to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the shader better.

Good prompt: Implement/inspect the shader in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Particle system

Plain meaning: A system generating many small visual elements for effects.

Programmer meaning: Game developers use particle system to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the particle system better.

Good prompt: Implement/inspect the particle system in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Animation state

Plain meaning: Current animation and transitions, such as idle/run/attack/death.

Programmer meaning: Game developers use animation state to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the animation state better.

Good prompt: Implement/inspect the animation state in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: AI behavior

Plain meaning: Rules or algorithms controlling non-player entities.

Programmer meaning: Game developers use ai behavior to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the ai behavior better.

Good prompt: Implement/inspect the ai behavior in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Difficulty curve

Plain meaning: How challenge changes over time.

Programmer meaning: Game developers use difficulty curve to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the difficulty curve better.

Good prompt: Implement/inspect the difficulty curve in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Juice

Plain meaning: Extra feedback that makes actions feel satisfying: shake, particles, sound, squash, timing.

Programmer meaning: Game developers use juice to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the juice better.

Good prompt: Implement/inspect the juice in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Game feel

Plain meaning: The subjective responsiveness and tactile quality of play.

Programmer meaning: Game developers use game feel to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the game feel better.

Good prompt: Implement/inspect the game feel in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Feedback loop

Plain meaning: How player action produces response, learning, adaptation, and reward/punishment.

Programmer meaning: Game developers use feedback loop to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the feedback loop better.

Good prompt: Implement/inspect the feedback loop in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Playtest

Plain meaning: A session where people or bots play to reveal design and technical issues.

Programmer meaning: Game developers use playtest to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the playtest better.

Good prompt: Implement/inspect the playtest in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Balance

Plain meaning: Tuning systems so challenge, fairness, and strategy feel right.

Programmer meaning: Game developers use balance to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the balance better.

Good prompt: Implement/inspect the balance in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Telemetry

Plain meaning: Runtime data collected to understand play behavior.

Programmer meaning: Game developers use telemetry to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the telemetry better.

Good prompt: Implement/inspect the telemetry in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: Spawn logic

Plain meaning: Rules for creating enemies, items, hazards, or players.

Programmer meaning: Game developers use spawn logic to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the spawn logic better.

Good prompt: Implement/inspect the spawn logic in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

Term: State machine

Plain meaning: A model of discrete states and transitions, like menu/play/dead/restart or idle/chase/attack.

Programmer meaning: Game developers use state machine to discuss a specific layer of play, simulation, feedback, or rendering.

Why Codex cares: Codex needs this word to place changes in the right part of the game rather than sprinkling code into whatever file looks dramatic.

Beginner misunderstanding: Beginners collapse mechanics, visuals, and feel into one vague request.

Bad prompt: Make the state machine better.

Good prompt: Implement/inspect the state machine in the existing game architecture. Define expected behavior, affected files, validation, and edge cases before editing.

How to prompt Codex for game work

- Build the simulation first. Placeholder visuals are acceptable until mechanics work.
- Separate mechanics from visuals. A beautiful dash that breaks collision is not a feature.
- Make state inspectable through debug overlays, logs, or testable data structures.
- Wire minimaps/debug views to true game state, not a parallel approximation.
- Add developer overlays for hitboxes, collision grid, entity IDs, AI targets, frame time, and spawn zones.
- Use playtest bots or automated QA where possible.
- Validate input, collisions, win/loss states, reset behavior, pause, and edge cases.
- Avoid polishing one wall while the arena, loop, and collisions are missing.

Building a Tron-style lightcycle prototype

```
Objective: Build a playable Tron-style lightcycle prototype.
Scope: Simulation first: grid movement, direction changes, trail creation, wall/trail collision, death, restart.
Do not: Add fancy art, networking, menus, or particle polish.
Acceptance criteria: Player can steer, leaves trail, dies on collision, restarts cleanly, no console errors.
Validation: Launch locally, play for 3 rounds, report controls, observed bugs, and changed files.
```

Adding arena hazards

```
Objective: Add timed arena hazards.
Scope: Add hazard entities tied to simulation state and debug overlay.
Rules: Hazards spawn in legal cells, telegraph for 1 second, then become lethal for 2 seconds.
Acceptance criteria: Player dies on active hazard, not during telegraph; hazard clears; tests cover timing.
Validation: Run game tests and playtest with hazard debug overlay.
```

Creating disappearing map chunks

> Objective: Add disappearing map chunks without breaking collision.
> Scope: Simulation grid owns chunk state; renderer visualizes it.
> Rules: Chunks warn visually, disappear, become non-traversable, then optionally restore.
> Acceptance criteria: True collision grid matches minimap/debug overlay; no desync after restart.
> Validation: Add tests for chunk state transitions and playtest.

Implementing a minimap tied to actual game state

> Objective: Add minimap based on real arena state.
> Scope: Read from simulation state only. Do not create a separate minimap model.
> Acceptance criteria: Player, trails, hazards, and missing chunks match actual state every frame.
> Validation: Add debug comparison or assertions; inspect screenshot.

Adding bot QA

> Objective: Add automated bot playtest harness.
> Scope: Bot should run deterministic rounds to stress collision/restart/hazard logic.
> Acceptance criteria: 100 simulated rounds with no crashes; logs summarize deaths and anomalies.
> Validation: Run harness and report seed, round count, failures.

Creating bug reports from playtesting

> Objective: Convert playtest observations into defect reports.
> Scope: Do not fix yet.
> Acceptance criteria: Each report has title, severity, repro steps, expected/observed behavior, evidence, suspected system, and recommended next pass.

Moving from primitive prototype to visual polish

> Objective: Polish visuals after mechanics pass.
> Scope: Improve sprites, effects, camera shake, HUD clarity, and audio cues.
> Do not: Change mechanics, collision, or timing unless explicitly approved.
> Acceptance criteria: Same simulation tests pass; visuals improve readability; no performance regression.

Part 13: Design, UI, and UX Language

Term: UX

Plain meaning: User experience: how it feels to accomplish a goal.

Programmer meaning: Designers use ux to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the ux nice.

Good prompt: Improve the ux for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: UI

Plain meaning: User interface: visible/interactable controls and surfaces.

Programmer meaning: Designers use ui to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the ui nice.

Good prompt: Improve the ui for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Layout

Plain meaning: Spatial arrangement of elements.

Programmer meaning: Designers use layout to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the layout nice.

Good prompt: Improve the layout for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Hierarchy

Plain meaning: Relative importance shown through size, position, spacing, contrast.

Programmer meaning: Designers use hierarchy to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the hierarchy nice.

Good prompt: Improve the hierarchy for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Affordance

Plain meaning: A visual cue suggesting how something can be used.

Programmer meaning: Designers use affordance to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the affordance nice.

Good prompt: Improve the affordance for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Feedback

Plain meaning: System response to user action.

Programmer meaning: Designers use feedback to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the feedback nice.

Good prompt: Improve the feedback for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: State

Plain meaning: The current condition of UI: selected, disabled, loading, error, empty.

Programmer meaning: Designers use state to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the state nice.

Good prompt: Improve the state for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Empty state

Plain meaning: What appears when no data exists.

Programmer meaning: Designers use empty state to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the empty state nice.

Good prompt: Improve the empty state for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Loading state

Plain meaning: What appears while work is pending.

Programmer meaning: Designers use loading state to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the loading state nice.

Good prompt: Improve the loading state for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Error state

Plain meaning: What appears when something fails.

Programmer meaning: Designers use error state to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the error state nice.

Good prompt: Improve the error state for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Responsive design

Plain meaning: Layout adapting to screen sizes/devices.

Programmer meaning: Designers use responsive design to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the responsive design nice.

Good prompt: Improve the responsive design for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Accessibility

Plain meaning: Making UI usable by people with varied abilities and assistive technologies.

Programmer meaning: Designers use accessibility to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the accessibility nice.

Good prompt: Improve the accessibility for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Contrast

Plain meaning: Visual difference between elements, often text/background.

Programmer meaning: Designers use contrast to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the contrast nice.

Good prompt: Improve the contrast for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Motion

Plain meaning: Animation and transitions.

Programmer meaning: Designers use motion to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the motion nice.

Good prompt: Improve the motion for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Interaction model

Plain meaning: Rules for how users act and receive responses.

Programmer meaning: Designers use interaction model to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the interaction model nice.

Good prompt: Improve the interaction model for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Information architecture

Plain meaning: Organization of content/navigation.

Programmer meaning: Designers use information architecture to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the information architecture nice.

Good prompt: Improve the information architecture for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: User flow

Plain meaning: Sequence of steps to accomplish a task.

Programmer meaning: Designers use user flow to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the user flow nice.

Good prompt: Improve the user flow for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Wireframe

Plain meaning: Low-fidelity layout plan.

Programmer meaning: Designers use wireframe to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the wireframe nice.

Good prompt: Improve the wireframe for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Mockup

Plain meaning: Static visual design with more detail.

Programmer meaning: Designers use mockup to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the mockup nice.

Good prompt: Improve the mockup for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Prototype

Plain meaning: Interactive or partially functional model.

Programmer meaning: Designers use prototype to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the prototype nice.

Good prompt: Improve the prototype for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

Term: Design system

Plain meaning: Reusable components, tokens, rules, and patterns.

Programmer meaning: Designers use design system to specify behavior, structure, clarity, and visual communication.

Why Codex cares: Codex needs this to distinguish functional state handling from cosmetic changes.

Beginner misunderstanding: Beginners say "make it look better" instead of naming behavior, hierarchy, constraints, and states.

Bad prompt: Make the design system nice.

Good prompt: Improve the design system for this component: define expected behavior, layout hierarchy, responsive constraints, states to handle, and validation screenshot sizes.

How a non-designer can give Codex usable UI direction

- Describe intended behavior: what the user is trying to do and what should happen.
- Describe layout hierarchy: what should be most visible, secondary, hidden, or grouped.
- Provide visual references or screenshots when available, but label what to copy and what not to copy.
- Specify constraints: breakpoints, accessibility, keyboard, theme, design system components, browser targets.
- Demand functional state handling: empty, loading, error, disabled, success, focus, hover, mobile.
- Separate visual polish from implementation correctness. First make it work. Then make it elegant. Civilization continues, somehow.

Part 14: Documentation and Repo Memory

Documentation is not decorative. It is repo memory. Threads compact, agents forget, humans misremember, and Slack disappears under seven jokes about lunch. The repo is the most reliable place to record durable facts: what changed, why, how to validate, what remains broken, and what the next pass should do.

Term: README

Plain meaning: Entry document explaining what the project is, how to run it, and key workflows.

Programmer meaning: Teams use readme to keep shared project knowledge durable.

Why Codex cares: Codex can read and update readme so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update readme to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: CHANGELOG

Plain meaning: Chronological record of notable changes.

Programmer meaning: Teams use changelog to keep shared project knowledge durable.

Why Codex cares: Codex can read and update changelog so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update changelog to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: TODO

Plain meaning: List of pending tasks. Dangerous when stale.

Programmer meaning: Teams use todo to keep shared project knowledge durable.

Why Codex cares: Codex can read and update todo so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update todo to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: ROADMAP

Plain meaning: Planned milestones and direction.

Programmer meaning: Teams use roadmap to keep shared project knowledge durable.

Why Codex cares: Codex can read and update roadmap so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update roadmap to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: CONTRIBUTING

Plain meaning: Rules for contributors: setup, tests, style, PR expectations.

Programmer meaning: Teams use contributing to keep shared project knowledge durable.

Why Codex cares: Codex can read and update contributing so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update contributing to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: Architecture notes

Plain meaning: Explanation of system structure and decisions.

Programmer meaning: Teams use architecture notes to keep shared project knowledge durable.

Why Codex cares: Codex can read and update architecture notes so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update architecture notes to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: Agent handoff notes

Plain meaning: Short current-state notes for the next agent/human.

Programmer meaning: Teams use agent handoff notes to keep shared project knowledge durable.

Why Codex cares: Codex can read and update agent handoff notes so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update agent handoff notes to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: Test plan

Plain meaning: What should be tested and how.

Programmer meaning: Teams use test plan to keep shared project knowledge durable.

Why Codex cares: Codex can read and update test plan so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update test plan to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: QA log

Plain meaning: Observed defects, checks, and evidence from QA passes.

Programmer meaning: Teams use qa log to keep shared project knowledge durable.

Why Codex cares: Codex can read and update qa log so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update qa log to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: Known issues

Plain meaning: Current problems and limitations.

Programmer meaning: Teams use known issues to keep shared project knowledge durable.

Why Codex cares: Codex can read and update known issues so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update known issues to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: Decision records

Plain meaning: Architecture Decision Records: what decision was made and why.

Programmer meaning: Teams use decision records to keep shared project knowledge durable.

Why Codex cares: Codex can read and update decision records so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update decision records to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Term: Release notes

Plain meaning: User/reviewer-facing summary of shipped changes.

Programmer meaning: Teams use release notes to keep shared project knowledge durable.

Why Codex cares: Codex can read and update release notes so future passes have accurate context.

Beginner misunderstanding: Beginners treat docs as marketing or homework. They are operational memory.

Bad prompt: Update docs.

Good prompt: Update release notes to match actual implementation. Remove stale claims and include validation results and remaining gaps.

Repo memory rules

- Record what changed.
- Record why it changed.
- Record what remains broken.
- Record validation results.
- Record next steps.
- Avoid stale docs.
- Remove obsolete claims.
- Keep handoff notes short but exact.

Agent handoff note

```
# Agent Handoff
Date:
Branch/worktree:
Objective:
Completed work:
Changed files:
Validation run:
Results:
Known issues:
Untested risks:
Recommended next pass:
Do not do next:
```

QA defect report

```
# QA Defect
Title:
Severity:
Area:
Environment:
```

```
Observed behavior:
Expected behavior:
Repro steps:
Evidence:
Suspected cause:
Workaround:
Recommended next action:
```

Implementation summary

```
# Implementation Summary
Objective:
Approach:
Files changed:
Behavior changed:
Behavior intentionally unchanged:
Validation:
Risks:
Follow-up:
```

Architecture decision record

```
# ADR: <Decision>
Status:
Context:
Decision:
Alternatives considered:
Consequences:
Validation/rollback:
```

Test plan

```
# Test Plan
Feature/system:
Unit tests:
Integration tests:
E2E/manual QA:
Performance/visual checks:
Edge cases:
Tools/commands:
Pass criteria:
Known gaps:
```

Release checklist

```
# Release Checklist
Branch/commit/tag:
Scope:
Build:
Tests:
Manual QA:
Security/dependency checks:
Docs updated:
Known issues:
Rollback plan:
Release notes:
Approval:
```

Part 15: Failure Modes and How to Correct Them

Overengineering

Symptoms: New abstractions, factories, managers, plugin systems, or generalized frameworks appear for a simple task.

Cause: Agent tries to impress, or prompt asks for "scalable" without real constraints.

How to detect it: Review diff for extra layers and unused extension points.

How to redirect Codex: Ask for the smallest local solution and removal of speculative abstractions.

Corrective prompt: Revert speculative abstractions. Implement the smallest solution that satisfies the acceptance criteria and explain why it is sufficient.

Under-specification

Symptoms: Agent asks many questions or makes broad assumptions; output mismatches intent.

Cause: Prompt lacks objective, scope, constraints, and done criteria.

How to detect it: Compare output to missing requirements; inspect assumptions section.

How to redirect Codex: Restate task with template and acceptance criteria.

Corrective prompt: Stop implementation. Ask me for missing requirements only if blocked; otherwise produce a scoped plan with assumptions clearly labeled.

Fake completion

Symptoms: Agent says done without tests/build/app launch evidence.

Cause: No evidence requirement or validation unavailable but hidden.

How to detect it: Look for exact commands, outputs, screenshots, logs.

How to redirect Codex: Demand validation or honest gap report.

Corrective prompt: Do not claim completion. Run validation or explain exactly why it cannot run, then list remaining risk.

Refactor drift

Symptoms: Diff touches unrelated files or changes behavior during bug fix.

Cause: Agent expands scope to improve surrounding code.

How to detect it: Review `git diff --stat` and files modified.

How to redirect Codex: Constrain to minimal fix; revert unrelated changes.

Corrective prompt: Revert unrelated refactor changes. Keep only changes required for this bug and report why each file remains modified.

Test avoidance

Symptoms: Agent writes code but does not run or add tests.

Cause: Prompt did not require tests; environment failure; time pressure.

How to detect it: Final report lacks test commands.

How to redirect Codex: Require smallest relevant tests and gap disclosure.

Corrective prompt: Run the smallest relevant test suite now. If tests cannot run, report blocker and propose a testable alternative.

Test theater

Symptoms: Tests pass but do not prove behavior; snapshots blindly updated; mocks assert implementation only.

Cause: Agent optimizes for green status.

How to detect it: Read tests for behavioral assertions and pre-fix failure.

How to redirect Codex: Ask for tests that fail before fix or cover user-visible behavior.

Corrective prompt: Revise tests to assert observable behavior. Explain whether the test would fail on the original bug.

Hallucinated APIs

Symptoms: Code calls methods/classes/options that do not exist.

Cause: Agent assumes API from memory or another version.

How to detect it: Typecheck/build fails; docs/version mismatch.

How to redirect Codex: Check installed version and official docs/types.

Corrective prompt: Verify the installed package/SDK version and replace nonexistent APIs with documented calls. Run typecheck/build.

Infinite polishing

Symptoms: Agent keeps refining visuals, names, or docs without completing core behavior.

Cause: No stop condition or priority order.

How to detect it: Long diff; core acceptance still unmet.

How to redirect Codex: Stop polish, return to acceptance criteria.

Corrective prompt: Stop polishing. Evaluate only the original acceptance criteria and implement the next smallest missing requirement.

Local optimum obsession

Symptoms: Agent repeatedly improves one component while system goal remains unmet.

Cause: Narrow tool feedback rewards local changes.

How to detect it: Progress notes mention same subsystem repeatedly.

How to redirect Codex: Reframe around end-to-end behavior.

Corrective prompt: Pause. Map the end-to-end flow and identify the system-level blocker before further local edits.

Breaking unrelated systems

Symptoms: A change in shared component breaks other pages/features.

Cause: Agent ignores shared usage.

How to detect it: Search call sites; run broader tests.

How to redirect Codex: Require caller impact analysis.

Corrective prompt: List all call sites of the changed component and run relevant tests/pages for the top-risk usages.

Silent behavior changes

Symptoms: Refactor changes output, timing, data shape, or defaults without saying so.

Cause: Agent thinks behavior change is harmless.

How to detect it: Compare before/after tests, snapshots, docs, configs.

How to redirect Codex: Ask for behavior delta report.

Corrective prompt: Audit the diff for behavior changes. Classify each as intentional, accidental, or uncertain; revert accidental ones.

Ignoring edge cases

Symptoms: Happy path works; null/empty/max/concurrent/reset fails.

Cause: Acceptance criteria omit boundaries.

How to detect it: Ask for edge-case list and tests.

How to redirect Codex: Add edge-case validation.

Corrective prompt: List edge cases for this feature and add tests for the highest-risk three.

Misreading user intent

Symptoms: Implementation technically works but solves wrong problem.

Cause: Prompt ambiguity or agent assumption.

How to detect it: Compare to product goal and expected user behavior.

How to redirect Codex: Restate design intent and ask for plan correction.

Corrective prompt: Stop coding. Restate the intended user experience and identify where the current implementation diverges.

Excessive abstraction

Symptoms: Similar to overengineering but specifically hides simple behavior behind layers.

Cause: Agent favors general patterns.

How to detect it: Diff shows generic names, interfaces with one implementation.

How to redirect Codex: Collapse unnecessary layers.

Corrective prompt: Simplify the design. Remove one-use abstractions unless they protect a real public boundary.

Deleting useful code

Symptoms: Agent removes code/assets/docs labeled unused but actually dynamically referenced.

Cause: No evidence requirement for deletion.

How to detect it: Search dynamic references, manifests, build output.

How to redirect Codex: Restore and require deletion proof.

Corrective prompt: Before deleting anything, provide evidence from imports, manifests, build config, runtime references, and tests. Restore any uncertain deletion.

Impressive but unmaintainable code

Symptoms: Clever, dense, undocumented, hard to test.

Cause: Agent optimizes for sophistication.

How to detect it: Can you explain it? Are tests simple?

How to redirect Codex: Ask for readable conventional rewrite.

Corrective prompt: Rewrite this for maintainability: simple names, local logic, tests, no clever shortcuts.

Docs that lie about implementation

Symptoms: README says feature works or command exists when code differs.

Cause: Agent documents intended state, not actual state.

How to detect it: Search docs against code; run commands.

How to redirect Codex: Require docs verification.

Corrective prompt: Audit docs against implementation and validation. Remove claims not supported by code or tests.

Failing to run the app

Symptoms: Build/tests run but UI/game crash or misrender.

Cause: Agent avoids manual/browser validation.

How to detect it: No launch evidence; console errors.

How to redirect Codex: Require launch or honest blocker.

Corrective prompt: Launch the app/game using documented command, inspect console/logs, and report whether the target flow works.

Failing to inspect screenshots/logs

Symptoms: Agent ignores provided visual/evidence context.

Cause: Prompt did not force evidence-first pass.

How to detect it: Output lacks reference to screenshot/log details.

How to redirect Codex: Ask for evidence analysis before edits.

Corrective prompt: Analyze the provided screenshot/log first. Identify exact clues and likely files before changing code.

Confusing visual similarity with functional correctness

Symptoms: Looks right but button, state, collision, or API broken.

Cause: Visual QA only.

How to detect it: Interact with UI/game and inspect state/logs.

How to redirect Codex: Pair visual check with behavior check.

Corrective prompt: After visual changes, interact with the control and verify state/API/collision behavior, not just appearance.

Part 16: The Orchestrator's Checklist Library

Starting a new project

- ☐ Define product goal
- ☐ Define target user/player
- ☐ Choose scope: prototype, MVP, vertical slice, production
- ☐ List constraints
- ☐ Choose stack intentionally
- ☐ Create repo structure
- ☐ Add README with run commands
- ☐ Add AGENTS.md with standards
- ☐ Create first milestone
- ☐ Define validation loop

Inspecting an unfamiliar repo

- ☐ Check Git status and branch
- ☐ Read README/AGENTS.md/package scripts
- ☐ Identify project type and entry points
- ☐ Map directories/modules
- ☐ Find build/test/lint commands
- ☐ Identify data model/state owners
- ☐ List dependencies/toolchain
- ☐ Run safe validation if appropriate
- ☐ Report architecture and risks
- ☐ Do not edit until plan approved

Asking Codex to build a feature

- ☐ Objective clear
- ☐ Acceptance criteria written
- ☐ Scope and non-goals listed
- ☐ Relevant files named
- ☐ Constraints stated
- ☐ Edge cases listed
- ☐ Validation commands provided
- ☐ Evidence required
- ☐ Stop conditions set
- ☐ Handoff requested

Asking Codex to fix a bug

- ☐ Observed behavior stated
- ☐ Expected behavior stated
- ☐ Repro steps provided
- ☐ Logs/screenshots attached
- ☐ Scope constrained
- ☐ Minimal fix required
- ☐ Regression test requested
- ☐ Validation commands listed
- ☐ Root cause required
- ☐ Untested risks required

Asking Codex to refactor

- ☐ Behavior preservation stated
- ☐ Public interfaces identified
- ☐ Current tests run first
- ☐ Plan before edits
- ☐ Small checkpoints
- ☐ No unrelated cleanup
- ☐ Diff review
- ☐ Tests after refactor
- ☐ Behavior delta audit
- ☐ Docs updated if architecture changes

Asking Codex to improve performance

- ☐ Baseline metric required
- ☐ Bottleneck identified
- ☐ One optimization at a time
- ☐ Correctness constraints stated
- ☐ Profiler/timing used
- ☐ Before/after compared
- ☐ No speculative rewrites
- ☐ Tests run
- ☐ Remaining bottlenecks listed

- ☐ Tradeoffs documented

Asking Codex to add tests

- ☐ Behavior specified
- ☐ Failure mode known
- ☐ Test level chosen
- ☐ Existing style followed
- ☐ Mocks justified
- ☐ Pre-fix failure considered
- ☐ Small relevant suite run
- ☐ Coverage gaps listed
- ☐ No production changes unless approved
- ☐ Test names readable

Asking Codex to perform QA

- ☐ Environment defined
- ☐ App/game launch command
- ☐ Flows listed
- ☐ Screenshots/logs required
- ☐ Repro steps for defects
- ☐ Severity assigned
- ☐ Known gaps listed
- ☐ No fixes unless requested
- ☐ QA log written
- ☐ Retest plan created

Reviewing Codex output

- ☐ Read diff
- ☐ Check modified file list
- ☐ Compare to acceptance criteria
- ☐ Check validation evidence
- ☐ Look for unrelated changes
- ☐ Look for dependency/config changes
- ☐ Look for deleted files
- ☐ Read tests for behavior
- ☐ Check docs for lies
- ☐ Ask what remains untested

Accepting a change

- ☐ Scope matches task
- ☐ Tests/build pass or gaps accepted
- ☐ No unrelated changes
- ☐ Risk acceptable
- ☐ Rollback exists
- ☐ Docs/handoff updated
- ☐ Commit message meaningful
- ☐ PR/review ready

- ☐ Known issues logged
- ☐ Next step clear

Rejecting a change

- ☐ Identify unacceptable files/behavior
- ☐ Preserve useful findings
- ☐ Ask to revert or narrow
- ☐ Restate acceptance criteria
- ☐ Require new plan
- ☐ Keep evidence
- ☐ Do not merge partial bad diff
- ☐ Create cleanup task if needed
- ☐ Document reason
- ☐ Restart from checkpoint

Preparing a release

- ☐ Git clean or intentional
- ☐ Version/tag ready
- ☐ Build passes
- ☐ Test suite passes
- ☐ Manual QA done
- ☐ Security/dependency check
- ☐ Docs/release notes updated
- ☐ Known issues listed
- ☐ Rollback plan ready
- ☐ Final approval recorded

Handing off to another agent

- ☐ Objective restated
- ☐ Branch/worktree named
- ☐ Changed files listed
- ☐ Completed work summarized
- ☐ Validation output included
- ☐ Failures/gaps listed
- ☐ Architecture notes linked
- ☐ Next pass exact
- ☐ Forbidden actions listed
- ☐ Stop conditions clear

Part 17: Prompt Library

Repo reconnaissance

> Inspect this repository before changing anything. Report Git status, project structure, entry points, build/test/lint commands, AGENTS.md guidance, major dependencies, current risks, and recommended first implementation pass. Do not edit files.

Architecture explanation

> Explain the architecture of this codebase/subsystem. Include responsibilities of major modules, data flow, state ownership, public interfaces, dependencies, and gotchas. Do not edit files.

Implementation planning

> Create an implementation plan for <feature/fix>. Include affected files, steps, risks, validation commands, acceptance criteria, rollback strategy, and what you need to confirm before editing. Do not implement yet.

Minimal bug fix

> Fix this bug with the smallest safe change. First reproduce or explain closest available evidence. Do not refactor unrelated code. Add a regression test if feasible. Run relevant validation and report evidence.

Feature build

> Implement <feature> according to these acceptance criteria: <criteria>. Scope is <files/subsystem>. Do not change <non-goals>. Run <validation>. Report modified files, validation, and untested risks.

Game mechanic build

> Add <mechanic> to the game simulation first, with placeholder visuals. Define state, input, collision/win/loss implications, tests, and debug visibility. Do not polish art until mechanics validate.

Visual polish pass

> Perform a visual polish pass only. Preserve behavior, mechanics, API, and timing. Use screenshots/preview to validate appearance. Report before/after visual changes and confirm tests/build still pass.

Performance profiling

> Measure current performance for <flow>. Identify top bottleneck with evidence. Propose one optimization. Implement only after confirming measurement method. Compare before/after and report metrics.

Test generation

> Add tests for <behavior>. Follow existing test style. Ensure tests assert observable behavior, not implementation trivia. Run the smallest relevant suite and report what remains untested.

QA playtest

> Launch the app/game and perform QA for these flows: <flows>. Capture logs/screenshots where useful. Do not fix bugs. Produce defect reports with repro steps, severity, expected/observed behavior, and evidence.

Screenshot-based visual correction

> Analyze the attached screenshot first. Identify visible defects, likely files, and constraints. Then make the smallest visual correction. Validate with a new screenshot or describe why screenshot capture is unavailable.

Diff review

> Review the current diff against the original objective and acceptance criteria. Flag unrelated changes, regressions, missing tests, risky patterns, config/dependency changes, and docs that may be stale. Do not edit files.

Security review

> Review this diff for security risks: secrets, auth/authorization, input validation, injection, dependency changes, logging of sensitive data, network calls, and unsafe file access. Report only high-confidence issues and evidence.

Dependency audit

> Inspect dependency changes. Explain why each new/updated dependency is needed, license/maintenance/security risks, lockfile impact, and whether existing dependencies could satisfy the need. Do not install anything without approval.

Documentation update

> Update documentation to match the actual implementation. Remove stale claims. Include run/test commands, changed architecture, known gaps, and validation results. Do not change code.

Agent handoff

> Write a concise handoff note for the next agent. Include objective, branch/worktree, changed files, completed work, validation results, failures, known risks, exact next steps, and actions to avoid.

Stop and explain before changing anything

> Stop. Do not edit files or run destructive commands. Explain your current understanding, assumptions, likely files, risks, and proposed next step. Wait for approval before implementation.

Undo risky changes

> Review the current diff and revert changes unrelated to <objective>. Preserve only necessary validated edits. Report reverted files and why. Do not add new behavior.

Find what broke

> Compare current behavior against the last known good state. Inspect recent diffs/commits, reproduce the failure, identify the first likely breaking change, and propose a minimal fix. Do not patch until root cause is stated.

Make the smallest safe fix

> Make the smallest safe fix for <issue>. No refactors, no new dependencies, no unrelated cleanup. Explain why each changed line/file is necessary. Validate and report evidence.

Write a plan but do not implement

> Write a plan only. Include affected files, steps, acceptance criteria, validation, risks, rollback, and open questions. Do not edit files.

Implement only this plan

> Implement only the approved plan below. Do not expand scope. If the plan is insufficient or risky, stop and explain rather than improvising. After implementation, run validation and report evidence.

Validate and report evidence

> Validate the current changes. Run the relevant commands, launch/inspect the app if needed, capture logs/screenshots where useful, and report exact evidence plus untested risks. Do not make new changes unless validation itself requires a tiny documented fix.

Part 18: Final Field Manual

10 core principles

1. Specify outcomes, not moods.
2. Separate inspect, plan, implement, validate, QA, cleanup, and docs.
3. Keep scope small and reviewable.
4. Use acceptance criteria for every serious task.
5. Demand evidence, not confidence.
6. Review diffs before accepting.
7. Preserve rollback points.
8. Make state, contracts, and data flow explicit.
9. Use repo memory for durable context.
10. Treat Codex as a capable teammate, not an unsupervised owner.

10 dangerous anti-patterns

1. "Make it better" prompts
2. Full-access unattended broad changes
3. Skipping Git status
4. Accepting "should work"
5. Mixing refactor with bug fix
6. Adding dependencies casually
7. Updating snapshots without behavioral review
8. Parallel agents on same files
9. Docs that describe intended rather than actual behavior
10. Polishing visuals before core loop works

10 phrases every code orchestrator should use

1. Do not edit yet. First inspect and explain.
2. Create a plan with affected files, risks, and validation.
3. Keep the fix minimal.
4. List every modified file and why it changed.
5. Run the smallest relevant test suite.
6. Report exact command output.
7. What did you not test?
8. Preserve public interfaces unless approved.
9. Stop before changing unrelated code.
10. Write a handoff note for the next pass.

10 phrases that cause chaos

1. Make it production quality.
2. Clean up whatever you see.

3. Use any library you want.
4. Just fix all bugs.
5. Make it more modern.
6. Refactor it while you are there.
7. Ship it.
8. It should be obvious.
9. Add some AI.
10. Make it fun.

One-page Codex task template

```
Objective:
Context:
Task type: Recon / Plan / Implement / Debug / Refactor / Test / QA / Docs / Cleanup / Handoff
Scope:
Non-goals:
Relevant files/directories:
Constraints:
Expected behavior:
Acceptance criteria:
Validation commands:
Evidence required:
Risk areas:
Permission/sandbox expectations:
Stop conditions:
Handoff/documentation requirement:
```

One-page review checklist

- ☐ Does the diff match the objective?
- ☐ Are modified files expected?
- ☐ Any unrelated cleanup/refactor?
- ☐ Any public interface/schema/config/dependency change?
- ☐ Do tests assert behavior?
- ☐ Were tests/build/app launched?
- ☐ Any docs updated incorrectly?
- ☐ Any edge cases missed?
- ☐ Any rollback point?
- ☐ Is handoff clear?

One-page QA checklist

- ☐ Environment recorded
- ☐ App/game launched
- ☐ Critical flow exercised
- ☐ Observed vs expected documented
- ☐ Screenshots/logs captured
- ☐ Console/errors checked
- ☐ State reset/retry checked
- ☐ Edge cases checked
- ☐ Defects severity-ranked
- ☐ Validation gaps listed

One-page game-development orchestration checklist

- ☐ Core loop defined
- ☐ Simulation state owns truth
- ☐ Input mapped and tested
- ☐ Collision/hitboxes debug-visible
- ☐ Win/loss/restart states validated
- ☐ Minimap/debug views tied to true state
- ☐ Bots/hazards deterministic where possible
- ☐ Performance/frame-time observed
- ☐ Playtest defects logged
- ☐ Visual polish separated from mechanics

Appendix A: Official Codex capability map versus practice standards

Topic	Official Codex documentation says	Orchestration standard in this manual
Codex CLI	Local terminal agent can inspect repositories, edit files, and run commands in the selected directory; supports interactive mode, slash commands, config flags, images, reviews, subagents, web search, cloud tasks, scripting, MCP, and approval modes.[28][29][30]	Use CLI for tight local loops, explicit validation, diff review, and controlled permissions.
Codex app	Desktop experience for parallel project threads with worktrees, automations, Git functionality, browser previews, review surfaces, skills/plugins, and artifact previews.[31]	Use app for multi-thread orchestration, local/worktree handoff, visual/browser QA, and review.
Codex web/cloud	Cloud agent can work in background/parallel in its own environment; connects to GitHub and can create PRs.[32]	Use cloud tasks for delegated work after repo setup and environment validation.
Sandboxing and approvals	Sandbox defines technical boundaries; approval policy controls when Codex must ask; local defaults restrict network and writes to workspace; modes include read-only, workspace-write, danger-full-access; approval policies include untrusted/on-request/never.[33][34]	Start with tight permissions; loosen only for known workflows; never use full access casually.

[28] OpenAI Codex CLI docs: https://developers.openai.com/codex/cli

[29] OpenAI Codex CLI features: https://developers.openai.com/codex/cli/features

[30] OpenAI Codex CLI slash commands: https://developers.openai.com/codex/cli/slash-commands

[31] OpenAI Codex app docs: https://developers.openai.com/codex/app

[32] OpenAI Codex web/cloud docs: https://developers.openai.com/codex/cloud

[33] OpenAI Codex sandbox docs: https://developers.openai.com/codex/concepts/sandboxing

[34] OpenAI Codex agent approvals and security: https://developers.openai.com/codex/agent-approvals-security

Topic	Official Codex documentation says	Orchestration standard in this manual
Cloud environments	Cloud tasks create containers, run setup, apply internet settings, run agent commands, and show diffs; setup has internet, agent internet is off by default unless enabled.[35][36]	Configure setup scripts and validation commands so cloud results are meaningful.
AGENTS.md	Codex reads AGENTS.md/override files before work and layers global/project/directory guidance with documented precedence and limits.[37]	Store durable repo rules, test commands, conventions, and do-not rules in AGENTS.md.
MCP	MCP connects Codex to third-party tools/context; Codex supports MCP servers in CLI and IDE extension, with STDIO and Streamable HTTP options.[38]	Use MCP only when it removes a real manual loop or provides current external context.
Skills	Skills package reusable instructions, resources, optional scripts; available in CLI, IDE, app; can be invoked explicitly or implicitly.[39]	Turn repeated prompts into scoped skills after the workflow works manually.
Automations	Codex app automations schedule recurring tasks, can run in local project or worktree, use default sandbox settings, and should be tested manually first.[40]	Automate only stable, reviewable workflows with clear stop/report conditions.
Subagents	Subagents are enabled by default, spawn only when explicitly asked, surface in app/CLI, inherit sandbox policy, and consume more tokens.[41]	Use subagents for bounded parallel exploration/review, not uncontrolled implementation fan-out.
Worktrees	Codex app worktrees use Git worktrees under the hood; they isolate parallel tasks and support handoff between local and worktree.[42]	Use worktrees for parallel agents, risky experiments, and background tasks.
GitHub code review	Configured Codex code review can respond to @codex review, review PR diffs, follow AGENTS.md guidance, and focus on serious P0/P1 issues; follow-up comments can ask Codex to fix issues.[43]	Use it as an extra review signal, never as sole merge approval.

[35] OpenAI Codex cloud environments: https://developers.openai.com/codex/cloud/environments

[36] OpenAI Codex cloud internet access: https://developers.openai.com/codex/cloud/internet-access

[37] OpenAI Codex AGENTS.md docs: https://developers.openai.com/codex/guides/agents-md

[38] OpenAI Codex MCP docs: https://developers.openai.com/codex/mcp

[39] OpenAI Codex skills docs: https://developers.openai.com/codex/skills

[40] OpenAI Codex app automations: https://developers.openai.com/codex/app/automations

[41] OpenAI Codex subagents docs: https://developers.openai.com/codex/subagents

[42] OpenAI Codex app worktrees: https://developers.openai.com/codex/app/worktrees

[43] OpenAI Codex GitHub code review docs: https://developers.openai.com/codex/integrations/github

Topic	Official Codex documentation says	Orchestration standard in this manual
In-app browser/game QA	Codex app browser use can operate local pages, click/type/inspect rendered state, take screenshots, and verify fixes when the Browser plugin is enabled.[44]	Use browser evidence for UI/game validation, paired with behavior and logs.
Game development use case	OpenAI recommends concrete game planning, PLAN.md, AGENTS.md, live browser testing, image generation prompts, and skills for browser-game workflows.[45]	Build playable mechanics first, then visual polish.

Appendix B: Footnotes and official source links

Official Codex, Git, and related documentation cited in this manual:

1. OpenAI Codex CLI docs: https://developers.openai.com/codex/cli
2. OpenAI Codex CLI features: https://developers.openai.com/codex/cli/features
3. OpenAI Codex CLI command line options: https://developers.openai.com/codex/cli/reference
4. OpenAI Codex CLI slash commands: https://developers.openai.com/codex/cli/slash-commands
5. OpenAI Codex app docs: https://developers.openai.com/codex/app
6. OpenAI Codex web/cloud docs: https://developers.openai.com/codex/cloud
7. OpenAI Codex best practices: https://developers.openai.com/codex/learn/best-practices
8. OpenAI Codex prompting docs: https://developers.openai.com/codex/prompting
9. OpenAI Codex sandbox docs: https://developers.openai.com/codex/concepts/sandboxing
10. OpenAI Codex agent approvals and security: https://developers.openai.com/codex/agent-approvals-security
11. OpenAI Codex AGENTS.md docs: https://developers.openai.com/codex/guides/agents-md
12. OpenAI Codex MCP docs: https://developers.openai.com/codex/mcp
13. OpenAI Codex skills docs: https://developers.openai.com/codex/skills
14. OpenAI Codex app automations: https://developers.openai.com/codex/app/automations
15. OpenAI Codex subagents docs: https://developers.openai.com/codex/subagents
16. OpenAI Codex app worktrees: https://developers.openai.com/codex/app/worktrees
17. OpenAI Codex cloud environments: https://developers.openai.com/codex/cloud/environments
18. OpenAI Codex cloud internet access: https://developers.openai.com/codex/cloud/internet-access
19. OpenAI Codex GitHub code review docs: https://developers.openai.com/codex/integrations/github
20. OpenAI Codex GitHub Action: https://developers.openai.com/codex/github-action

[44] OpenAI Codex app in-app browser docs: https://developers.openai.com/codex/app/browser

[45] OpenAI Codex create browser-based games use case: https://developers.openai.com/codex/use-cases/browser-games

21. OpenAI Codex app in-app browser docs: https://developers.openai.com/codex/app/browser

22. OpenAI Codex create browser-based games use case: https://developers.openai.com/codex/use-cases/browser-games

23. OpenAI engineering post, Unrolling the Codex agent loop: https://openai.com/index/unrolling-the-codex-agent-loop/

24. Official Git worktree docs: https://git-scm.com/docs/git-worktree

25. Official Git site: https://git-scm.com/

26. GitHub Docs, pull request merges: https://docs.github.com/articles/about-pull-request-merges